Разработка калькулятора на микроконтроллере с нуля

[https://github.com/igor-240340/HardwareCalculatorFromScratch]

[igor.240340@gmail.com]

Введение

Калькулятор будет работать под управлением AVR-микроконтроллера ATmega328P. Вводимые числа будут представлены в памяти калькулятора в формате бинарной палавающей точки одинарной точности (читай — сишные переменные типа float).

Поскольку весь функционал реализуется программно, а из встроенных арифметических инструкций используется только 8-битное целочисленное сложение, которое, наверное, можно найти в любом МК или процессоре, то за основу можно взять МК любой архитектуры и адаптировать код прошивки под него. Но если вы новичок, я бы рекомендовал для простоты и плавности процесса использовать то, что использует автор. Разобравшись один раз, вы потом сможете реализовать подобный калькулятор на чём угодно, хотя на «рассыпухе».

NOTE: Несмотря на наличие в AVR целочисленных инструкций умножения и, кажется, деления, а также вычитания, мы их не используем и реализуем всю базовую целочисленную арифметку с нуля, используя, по сути, только 8-битный сумматор.

Многие простые 8-разрядные микроконтроллеры не реализуют плавающую точку аппаратно, то есть, не имеют ни спец. регистров (как серьезные процессоры типа десктопных) ни инструкций, поэтому для реализации необходимо эмулировать плавающую точку программно. А поскольку внутреннее представление чисел — двоичное, а вводимые числа десятичные, то нам также потребуются подпрограммы конвертации. Весь код будет написан на ассемблере. Готовые библиотеки использоваться не будут.

NOTE: Коммерческие калькуляторы, как правило, работают с числами в двоично-десятичном формате (BCD – binary-coded decimals). То есть, что пользователь ввёл в калькулятор, над тем и будут произведеным арифметические операции, поэтому вводимые пользователем значения не требуют конвертации и, соответственно, не искажаются на этапе ввода. Более того, получаемые результаты вычислений также выводятся без конвертации и не искажаются на этапе вывода. Но, конечно, результат в BCD может быть достаточно большим и потребуется округления, но это будет единственный источник искажения.

Из чего будет состоять калькулятор? Во-первых, из аппаратной части и программной.

Аппаратная часть будет включать в себя:

- Микроконтроллер — здесь будет зашит код, который будет обрабатывать ввод/вывод от пользователя, конвертировать и вычислять.

- Энкодер клавиатуры — микросхема, к которой будет подключена 16-клавишная клавиатура, и которая будет выдавать на выходе 4-битный код клавиши, вместо 8-битного. Это избавит нас от необходимости реализовывать программно опрос клавиатуры и сэкономит количество входов на МК.
- LCD-дисплей, двухстрочный, 16-цифр. Сюда будут выводиться вводимые числа и выводиться результаты.
- Электротехнический обвес: резисторы, конденсаторы, стабилизатор, батарейка и пр. компоненты.

Программная часть будет включать в себя:

- Библиотека плавающей точки:
  - Подпрограммы FADD/FSUB для сложения/вычитания.
  - Подпрограмма FMUL для умножения.
  - Подпрограмма FDIV для деления.
  - Подпрограмма ATOF для конвертации введённой числовой строки в двоичное представление в формате плавающей точки.
  - Подпрограмма FTOA для конвертации результатов вычислений в числовую строку.
- Библиотека для взаимодействия с LCD-дисплеем:
  - Вывод одиночного символа.
  - Вывода строки.
  - Управления курсором (перемещение в конец, в начало, на первую или вторую строку).
  - Подпрограмма для очистки дисплея и сброса положения курсора в дефолтное.
- Код обработки нажатых клавиш. Это основной управляющий код калькулятора, здесь калькулятор понимает, что именно вводится: первый операнд, второй операнд, знак оператора или команда на вычисление. Здесь же обеспечивается корректный ввод — мы не должны дать пользователю ввести некорректную числовую строку, скажем, что-то вроде -0.5.6--..

Таким образом, структурно калькулятор будет выглядеть так, как показано на диаграмме ниже:

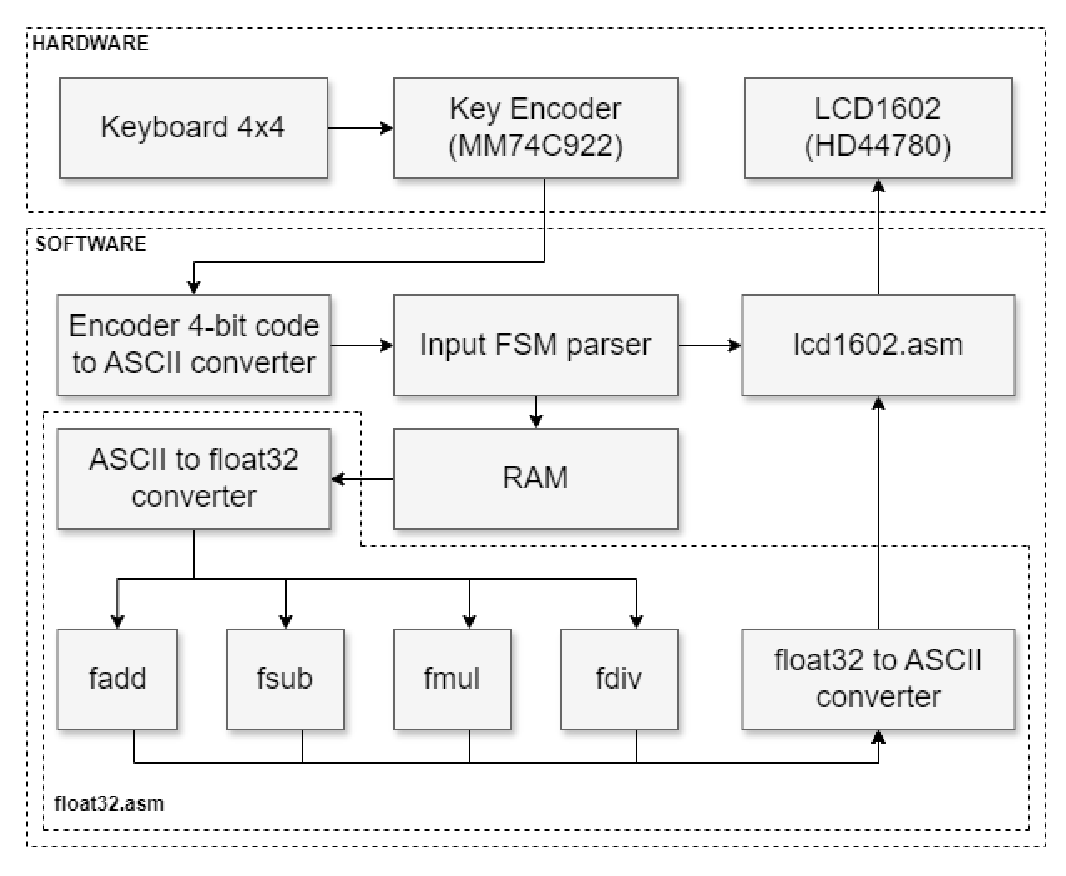


Рисунок 1

Все вводимые числа будут представлены в памяти как обычные ASCII-строки, оканчивающиеся нулём, и результат вычислений (после конвертации) также будет ASCII-строкой.

Идти будем по порядку: сначала разработаем библиотеку эмуляции плавающей точки и научимся выполнять вычисления над числами, которые изначально зашиты в память в двоичном виде, будто бы мы уже обработали ввод от пользователя и выполнили конвертацию. Затем зашьем в память числовые ASCII-строки и напишем конвертацию atof и ftoa, будто бы пользователь уже ввёл значения в память. Ну и наконец, реализуем ввод числовых строк с клавиатуры и вывод числовой строки результата на LCD.

Таким образом, дальнейший план будет состоять из следующих пунктов:

1. Разработка библиотеки эмуляции плавающей точки:
   1. Разработка fmul.
   2. Разработка fdiv.
   3. Разработка fadd/fsub.
2. Разработка подпрограммы atof.

3. Разработка подпрограммы ftoa.
4. Реализация ввода с клавиатуры.
5. Реализация вывод на LCD.

Конечный результат будет выглядеть примерно так:



Рисунок 2

Основы машинной арифметики

Представление целых чисел

Возьмем двоичную разрядную сетку длины 8 и свяжем с каждым разрядом индекс k справа налево начиная с нуля.

Индекс k	7	6	5	4	3	2	1	0
Разряд d	0	0	0	0	0	0	1	0

Рисунок 3

Значение, которое содержится в разряде будем обозначать через d_k, где k – индекс разряда. То есть, возьмем, например, разряд с индексом 1, который сейчас содержит единицу, тогда в любом месте в тексте мы будем обозначать это значение как $d_1 = 1$.

Теперь для каждого разряда с индексом k зададим вес w_k, значение которого будет определяться так:

$$w_k = d_k * 2^k, где\ k - индекс\ разряда. \quad (1)$$

Снова возьмем разряд с индексом 1, его вес равен:

$$w_1 = d_1 * 2^1 = 2, где\ d_1 = 1 - значение\ разряда\ с\ индексом\ 1. \quad (2)$$

Понятно, что если разряд содержит ноль, то его вес всегда будет равен нулю, независимо от положения в разрядной сетке.

Получается, что при такой схеме рассмотрения каждая отдельная двоичная единица в зависимости от своей позиции в разрядной сетке однозначно определяет какое-то десятичное число.

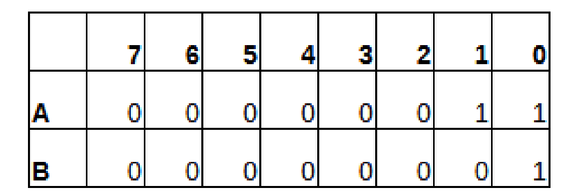
А чтобы определить, какое число представляет собой цепочка таких двоичных единиц, расположенных рядом, нужно просто сложить их весовые коэффициенты:

Индекс k	7	6	5	4	3	2	1	0
Разряд d	0	0	0	0	0	1	1	0
Вес	0	0	0	0	0	4	2	0
Σ весов	6							

Рисунок 4

Сложение целых чисел

Снова возьмем разрядую сетку длиной в 8 байт и представим в ней два числа $A=3$ и $B=1$.



	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	1	1
B	0	0	0	0	0	0	0	1

Рисунок 5

Теперь вычислим сумму этих чисел, но сначала в десятичной системе: $C=A+B=4$. Запишем этот результат в двоичном виде — это будет двоичная единица с индексом 2, поскольку её весовой коэффициент равен $1*2^2=4$.

	7	6	5	4	3	2	1	0
C	0	0	0	0	0	1	0	0

Рисунок 6

А теперь вычислим эту же сумму, но уже по правилам двоичной арифметики, т. е. не переходя к десятичной системе и обратно. Но сначала обратим внимание, что число A образовано двумя двоичными единицами, веса которых равны $2^0=1$ и $2^1=2$ соответственно, само же число, которое представлено этой двоичной цепочкой, равно сумме весовых коэффициентов, т. е. 1+2=3. Поэтому для начала разложим число A на два числа $A=A_0+A_1=1+2=3$.

	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	1	1
A_0	0	0	0	0	0	0	0	1
A_1	0	0	0	0	0	0	1	0

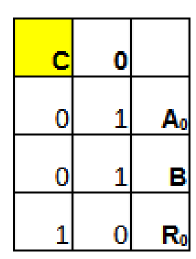
Рисунок 7

Понятно, что $A+B=3+1=4=(1+2)+1=A_0+A_1+B=1+2+1=4$. Поэтому давайте сначала сложим $A_0=1$ и $B=1$. Сумма равна двум, а в двоичном виде эта сумма однозначно может быть представлена двоичной единицей в позиции 1, т. к. её вес равен $2^1=2$.

	7	6	5	4	3	2	1	0
A_0	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	0	1
R_0	0	0	0	0	0	0	1	0

Рисунок 8

А теперь удалим разряды с индексами от 7 до 2 включительно и оставим только два индекса: 0 и 1, а буквы A_0, B и R_0 перенесём в правую часть. А еще, в строке индексов заменим единицу на С и выделим этот разряд желтым цветом.



C	0	
0	1	A_0
0	1	B
1	0	R_0

Рисунок 9

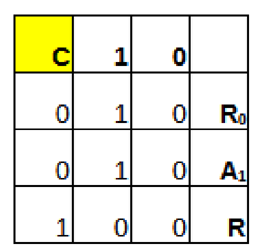
Верхняя строчка — это по-прежнему индексы двоичных разрядов, нумерация которых идет справа налево начиная с нуля, просто мы убрали лишние нулевые разряды. А разряд, помеченный латинской буквой C и выделенный желтым цветом — это тоже самое, что разряд с индексом 1, т. е. любая двоичная единица в этой позиции по-прежнему имеет вес $2^1=2$, просто мы дали этому разряду особый статус и выделили его цветом и буквой. Этот разряд называется разрядом переноса и сейчас станет ясно почему. Еще раз напомню, что мы взяли две единицы и сложили их, получив число 2, которое в двоичной системе можно обозначить единицей с индексом 1, т. к. её вес равен $2^1=2$. Но можно усмотреть в этом и общую закономерность — когда оба разряда в текущей позиции равны единице, разряд результата в этой позиции зануляется, а единица переносится в следующий разряд слева. И это справедливо для единиц в любой позиции в двоичной разрядной сетке. Если же только один разряд нулевой, то разряд результата будет равен единице: 0+1=1+0=1, ну а 0+0=0 — это и так понятно.

Хорошо, мы сложили A_0 и B и получили частичную сумму равную двум $R_0=2^1=2$. Теперь сложим эту частичную сумму с A_1. Мы уже знаем, что результат будет равен 4 и что в двоичном виде он может быть представлен двоичной единицей с индексом 2, вес которой равен $1*2^2=4$.

	7	6	5	4	3	2	1	0
R_0	0	0	0	0	0	0	1	0
A_1	0	0	0	0	0	0	1	0
R	0	0	0	0	0	1	0	0

Рисунок 10

И снова уберём нулевые разряды, перенесём буквы вправо, а в строке индексом разрядов 2 заменим на C.



C	1	0	
0	1	0	R_0
0	1	0	A_1
1	0	0	R

Рисунок 11

И снова мы видим, что справедлива та же закономерность, о которой мы выше писали — мы взяли разряды с индексом 0, сложили их и получили нулевой разряд результата в позиции 0 и нулевой разряд переноса. Затем мы взяли разряды с индексом 1, а т.к. оба они равны единице, то мы получили ноль в разряде результата в этой позиции и бит переноса в следующий разряд.

А теперь вернемся в полную разрядную сетку и посмотрим на процесс сложения еще раз.

	7	6	5	4	3	2	1	0
C							1	
A	0	0	0	0	0	0	1	1
B	0	0	0	0	0	0	0	1
R	0	0	0	0	0	1	0	0

Рисунок 12

Мы взяли разряды с индексом 0 чисел A и B и сложили их, получив бит переноса в соседний разряд. Затем мы взяли разряды с индексом 1 этих же чисел и сложили их и полученный бит переноса и снова получили бит переноса, который сложили уже с нулевыми разрядами A и B в позиции 2 и записали сразу в разряд результата. Так и работает двоичное сложение.

Вычитание целых чисел

Как и прежде, будем работать в разрядной сетке в пределах байта, но дополнительно покажем разряд бита переноса и рассмотрим два числа $A=3$ и $A'=253$:

	C	7	6	5	4	3	2	1	0
A		0	0	0	0	0	0	1	1
A'		1	1	1	1	1	1	0	1

Рисунок 13

А теперь сложим эти два числа:

	C	7	6	5	4	3	2	1	0
A		0	0	0	0	0	0	1	1
A'		1	1	1	1	1	1	0	1
R	1	0	0	0	0	0	0	0	0

Рисунок 14

Мы получили число 256, которое в двоичном виде представлено единицей с индексом 8: $2^8 = 256$. Мы также получили переполнение однобайтовой разрядной сетки. При этом, если мы проигнорируем бит переноса *C*, то значение, которое мы получили в пределах байта, равно нулю.

А теперь возьмем число $A=10$ и число $B=A'=253$ и вычислим их сумму:

	C	7	6	5	4	3	2	1	0
A		0	0	0	0	1	0	1	0
B=A'		1	1	1	1	1	1	0	1
R	1	0	0	0	0	0	1	1	1

Рисунок 15

Результат равен 263, и мы снова получили переполнение. Но если мы отбросим бит переноса и посмотрим только на сетку в пределах байта, то результат окажется равен 7. Такой же результат мы получим если от 10 отнимем 3 в десятичной системе.

Последние два примера показывают, что число 253 в пределах байта «ведёт» себя как -3.

Число 253 является дополнительным кодом числа -3 или дополнением до переполнения разрядной сетки, в данном случае — до $2^8=256$. В литературе такой способ представления отрицательных чисел известен как дополнение до двух (two's complement).

В общем случае, если у нас есть разрядная сетка длины *n*, то для получения доп. кода отрицательного числа нужно от наименьшего числа, которое уже не представимо в сетке, отнять модуль отрицательного: $2^n-|A|$.

На уровне реализации вычисление доп. кода оказывается еще проще, достаточно инвертировать биты числа (доп. код отрицательного значения которого мы хотим получить) и прибавить единицу. Рассмотрим вычисление доп. кода числа -3:

	C	7	6	5	4	3	2	1	0
A		0	0	0	0	0	0	1	1
INV(A)		1	1	1	1	1	1	0	0
INV(A)+1		1	1	1	1	1	1	0	1

Рисунок 16

Почему это работает? Давайте разберемся:

	C	7	6	5	4	3	2	1	0
2^8=256	1	0	0	0	0	0	0	0	0
256-1=255		1	1	1	1	1	1	1	1
|A|=3		0	0	0	0	0	0	1	1
255-|A|=252		1	1	1	1	1	1	0	0
1									1
252+1		1	1	1	1	1	1	0	1

Рисунок 17

Сначала мы предварительно отнимаем единицу от 256, получая все единицы в пределах байта или 255. Затем мы отнимаем от 255 модуль отрицательного числа $|-3|=3$ и получаем по факту инверсию битов числа 3. При этом, от исходного числа 256 мы суммарно отняли $3+1$, то есть на единицу больше, чем нужно для вычисления доп. кода в десятичной системе, поэтому прибавляем единицу обратно и получаем истинное значение доп. кода числа -3.

Алгебраически это можно записать так:

$$A'=2^n-|A|=[(2^n-1)-|A|]+1=INV(|A|)+1, \quad \text{где } INV-\text{инверсия битов двоичного представления числа } |A|. \tag{3}$$

Ну и для полноты покажем, что сложение в доп. коде действительно работает как вычитание. Допустим, что число $A>0$, а число $B<0$, тогда их разность $C=A+B, где\ B<0$. Но вместо разности рассмотрим сумму A с доп. кодом $B'=2^n-|B|$:

$$A+(2^n-|B|)=A+2^n-|B|=A+2^n+B=2^n+A+B, \text{ где } A+B=C-\text{это истинная разность}. \tag{4}$$

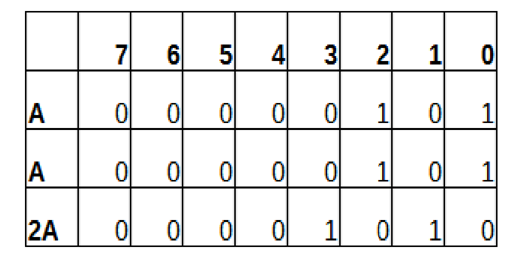
Мы видим, что если отбросить бит переноса, представленный здесь как 2^n, то мы действительно получим истинную разность равную $A+B, где\ B<0$.

TODO: Показать, что не все отрицательные значения представимы в пределах байта и сказать, что в некоторых случаях вычислять доп. код в двойной сетке нет необходимости. При этом обосновать разделения положительных и отрицательных значений в пределах сетки можно через требования к однозначному различению положительных и отрицательных между собой (MSB равный единице для отрицательных).

TODO: Добавить иллюстрацию механики доп. кода на циферблате.

Умножение целых чисел

Рассмотрим произведение: $5*2=10$. Оно эквивалентно сложению числа 5 с самим собой. При этом, как мы выше разбирали, из всех разрядов, где оба значения равны единице, возникает бит переноса, а разряд результата зануляется.



	7	6	5	4	3	2	1	0
A	0	0	0	0	0	1	0	1
A	0	0	0	0	0	1	0	1
2A	0	0	0	0	1	0	1	0

Рисунок 18

Хорошо видно, что если теперь этот результат умножить еще один раз на 2, то история повторится, и бит переноса появится разрядах с весами 2^2 и 2^4, что даст 20.

	7	6	5	4	3	2	1	0
2A	0	0	0	0	1	0	1	0
2A	0	0	0	0	1	0	1	0
4A	0	0	0	1	0	1	0	0

Рисунок 19

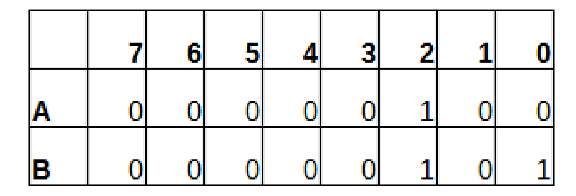
Это и есть истинный результат умножения числа 5 на 2 два раза подряд: $5*2*2=20$. При этом, если внимательно посмотрим на финальное произведение, то увидим, что результат умножения на 4=2*2 (т. е. два раза на два) это исходное число 5 сдвинутое влево два раза.

То есть, мы обнаружили еще одну закономерность: если у нас есть число в двоичной разрядной сетке, то для того, чтобы вычислить его произведение со степенью двойки, достаточно сдвинуть его влево на показатель степени двойки:

$$A*2^k=LSHIFT(A,k), где\ LSHIFT-операция\ сдвига\ двоичных\ разрядов\ числа\ A\ влево\ k \quad (5)$$

Базовая схема

Возьмем два числа в пределах байта: $A=4$ и $B=5$.



	7	6	5	4	3	2	1	0
A	0	0	0	0	0	1	0	0
B	0	0	0	0	0	1	0	1

Рисунок 20

Теперь разложим число B в сумму весовых коэффициентов его двоичных разрядов:

$$B=1*2^2+0*2^1+1*2^0=2^2+2^0, где\ 2^0=1. \quad (6)$$

А теперь запишем произведение $A*B$ с учетом этого разложения:

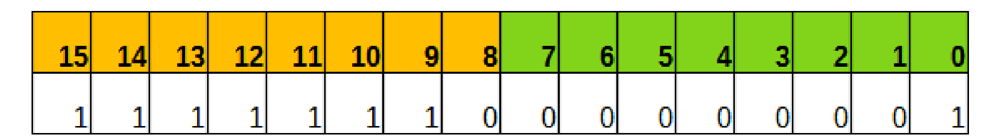
$$A*B=A*(2^2+1)=A*2^2+A*1=4*2^2+4*1=20. \quad (7)$$

Видим, что произведение чисел A и B можно вычислить как сумму частичных произведений числа A с весовыми коэффициентами ненулевых разрядов числа B. При этом помним, что весовые коэффициенты разрядов — это степени двойки, а чтобы вычислить произведение числа A со степенью двойки, как мы выше только что рассмотрели, достаточно двоичное представление числа A сдвинуть влево на показатель степени.

Здесь уже проглядывает простейший алгоритм для умножения чисел в двоичном виде: перебираем все двоичные разряды числа B начиная с младшего, и на каждом разряде (кроме самого первого) делаем сдвиг числа A влево — это даст нам на каждом шаге частичное произведение числа A с весовым коэффициентом текущего двоичного разряда числа B (в предположении, что разряд B равен единице). Далее остается только посмотреть, действительно ли текущий разряд B равен единице, если равен, то мы прибавляем сдвинутое число A к аккумулятору, если нет, то игнорируем и переходим к следующему разряду числа B.

А теперь перемножим два максимальных числа, представимых в пределах байта: 255*255. Произведение в десятичной системе равно 65025.

Но в двоичной системе этот результат уже не вмещается в байт и требует двух байт (но не больше):



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1

Рисунок 21

Действительно, возьмем два максимальных числа в пределах разрядной сетки длины *n* (при этом *n* будем полагать больше единицы) и запишем их произведение:

$$(2^n-1)*(2^n-1)=2^{2n}-2^{n+1}+1. \quad (8)$$

Величина $2^{2n}-1$ — это максимальное число в пределах сетки двойного размера, но из выражения 8 следует, что $2^{2n}-2^{n+1}+1=2^{2n}-(2^{n+1}-1)<(2^{2n}-1)\,,\mathit{для\,всех}\; n>1$, поэтому при перемножении чисел в пределах сетки размера *n* произведение в худшем случае потребует сетки двойного размера.

TODO: Привести блок-схему и пример кода на ассемблере.

TODO: Привести демонстрацию вычисления произведения в таблице (по аналогии со схемой с неподвижным множимым).

Схема с неподвижным множимым

Возьмем два числа: $A=4$ и $B=5$.

	7	6	5	4	3	2	1	0
A	0	0	0	0	0	1	0	0
B	0	0	0	0	0	1	0	1

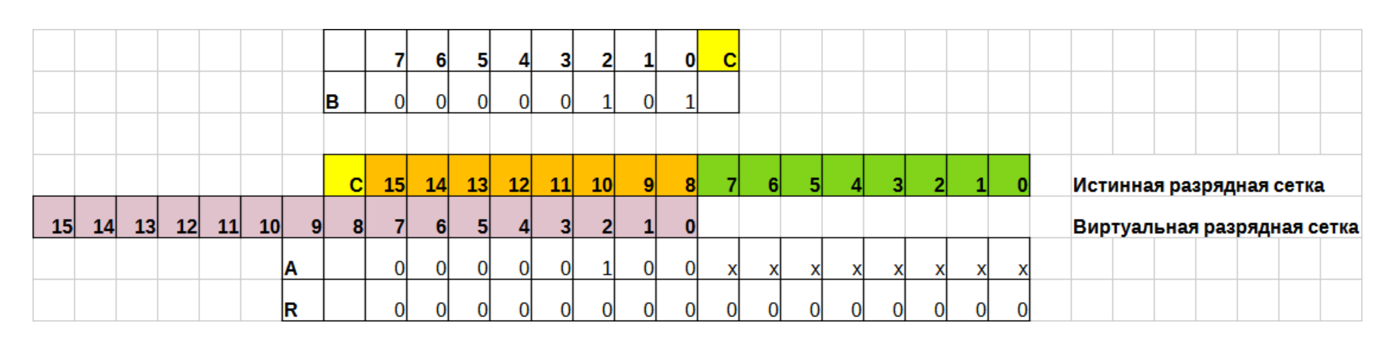
Рисунок 22

Множитель B разместим в пределах байта и будем сдвигать вправо, извлекая его разряды в бит переноса.

Под результат-аккумулятор R (там будет накапливаться сумма частичных произведений) отведём два байта (выше мы разобрали, почему два). Множимое A мысленно разместим в старшем байте двойной сетки (но фактически, на уровне реализации, под множимое отведем только один байт).

Свяжем теперь мысленно с неподвижным множимым A и аккумулятором R виртуальную разрядную сетку (фиолетовая на рисунке) двойного размера и разместим её так, чтобы A занимал младший байт в этой сетке.

В голове должна сложиться такая картина:



								7	6	5	4	3	2	1	0	C								
							B	0	0	0	0	0	1	0	1									
							C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Истинная разрядная сетка
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									Виртуальная разрядная сетка
						A		0	0	0	0	0	1	0	0	x	x	x	x	x	x	x	x	
						R		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Рисунок 23

Хотя в истинной двойной разрядной сетке результата R множимое A равно $A*2^8=4*2^8=1024$, относительно виртуальной сетки оно сохранило свое исходное значение равно 4.

Теперь сдвинем самый младший бит множителя (LSB – Least Signigicant Bit) в бит переноса. Он равен единице, его вес тоже равен единице: $1*2^0=1$. Частичное произведение числа A с весом этого бита равно своему исходному значению (его мы сейчас и имеем): $4*1=4$. Теперь нужно сложить это частичное произведение с текущим значением аккумулятора результата R. А поскольку младший байт аккумулятора содержит нули, складываем число A только со старшим байтом аккумулятора с записью суммы в этот же аккумулятор. Еще, в данном случае бит переноса оказался равен нулю.

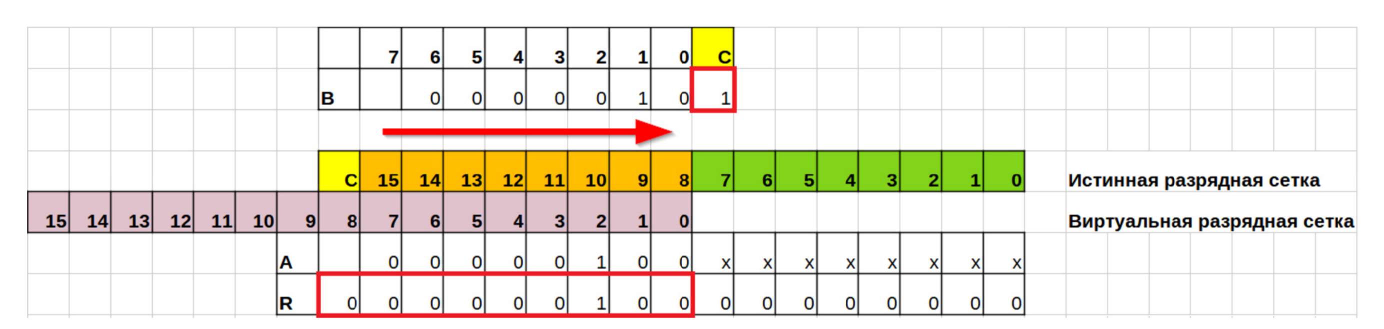


Рисунок 24

В аккумуляторе имеем теперь первое частичное произведение. Но прежде чем перейдем к следующей итерации, сдвинем аккумулятор вправо на один разряд вместе с виртуальной разрядной сеткой.

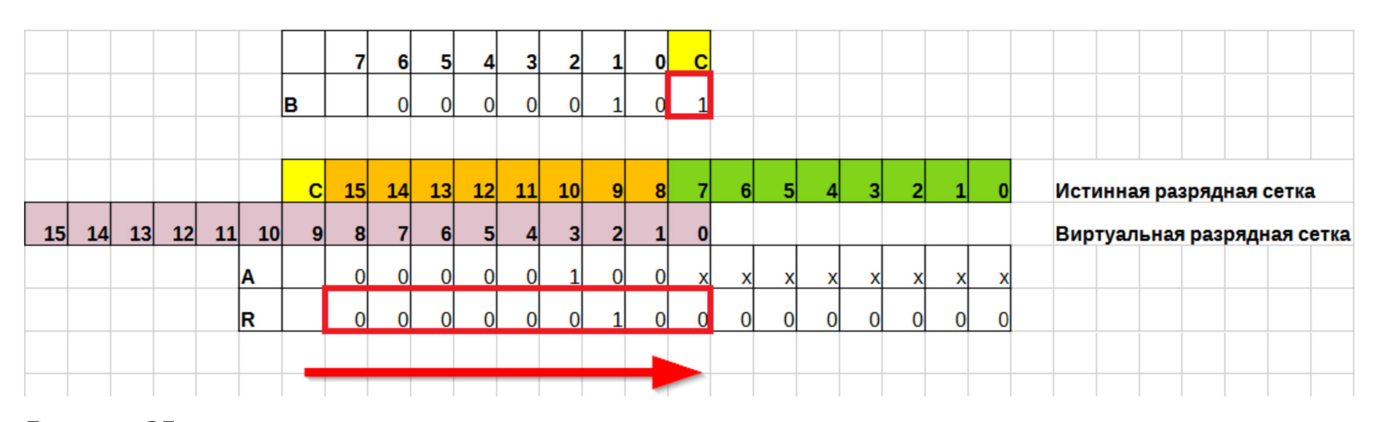


Рисунок 25

Как видно, это не изменило значения аккумулятора, но дало нам произведение множимого с весовым коэффициентом следующего бита, который мы будем извлекать, на случай, если он ненулевой: $A*2^1=4*2=8$.

А теперь снова сдвигаем множитель вправо, извлекая следующий бит.

	7	6	5	4	3	2	1	0	C	
B			0	0	0	0	0	1	0	1

Рисунок 26

Он нулевой, его вес равен $0*2^1=0$. Но если бы он был равен единице, то его вес был бы равен $1*2^1=2$ (а поскольку в конце последней итерации мы сдвинули аккумулятор R вместе с виртуальной рязрядной сеткой вправо, на текущей итерации неподвижное множимое A относительно подвижной виртуальной сетки уже представляет собой второе частичное произведение с весовым коэффициентом второго разряда (если бы он был равен единице). Но разряд нулевой, частичное произведение тоже нулевое, поэтому пропускаем сложение.

Теперь заглянем немного вперед, на третий шаг алгоритма, и предварительно посмотрим на следующий бит: он равен единице и его вес равен $1*2^2=4$. Это значит, что на следующем шаге частичное произведение будет равно $A*2^2=A*4=16$ и его нужно будет прибавить к аккумулятору.

Именно поэтому в конце каждой итерации, даже если текущий разряд оказался нулевым и мы пропустили сложение (как это было сейчас со вторым разрядом) мы все-равно сдвигаем аккумулятор вправо на один разряд вместе с виртуальной разрядной сеткой. Это мы и делаем после второго шага:

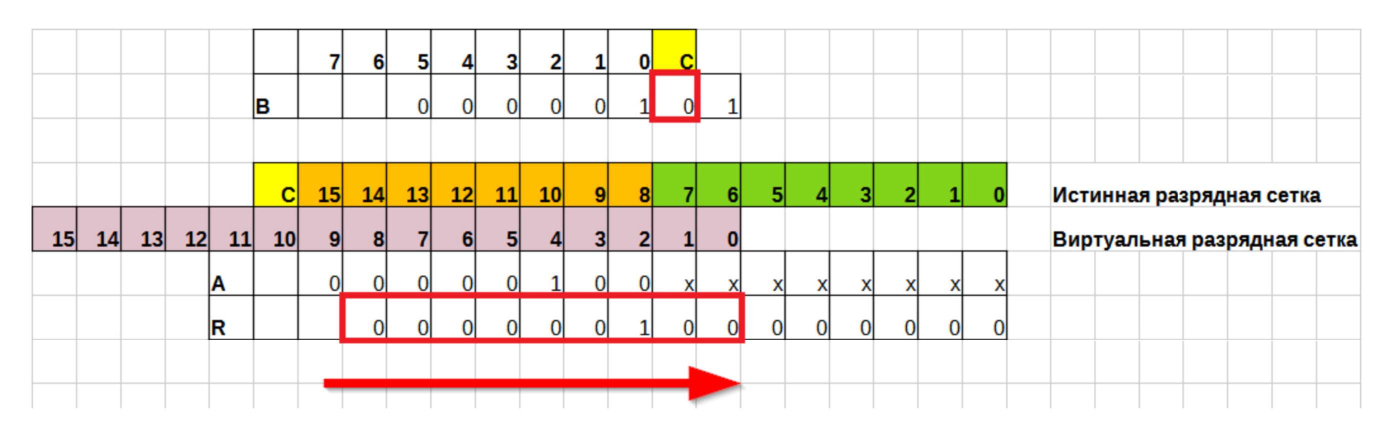


Рисунок 27

Как видим, в пределах виртуальной разрядной сетки, результат аккумулятора после второго сдвига содержит прежнее значение — первое частичное произведение, но множимое A при этом стало равным третьему частичному произведению (относительно виртуальной разрядной сетки), которое как раз потребуется нам на третьем шаге алгоритма.

Снова сдвигаем множитель вправо.

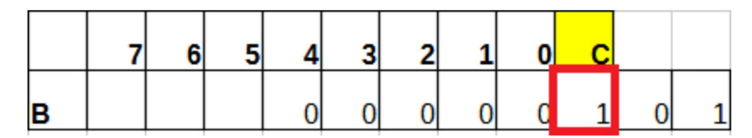


Рисунок 28

Как мы уже подметели выше — очередной разряд равен единице и его вес равен $1*2^2=4$, а это значит, что частичное произведение равно $A*4=4*4=16$. Но именно это частичное произведение и представляет текущее значение неподвижного множимого A относительно

виртуальной сетки после сдвига аккумулятора (вместе с сеткой) вправо на предыдущей итерации. Прибавляем A к аккумулятору R.

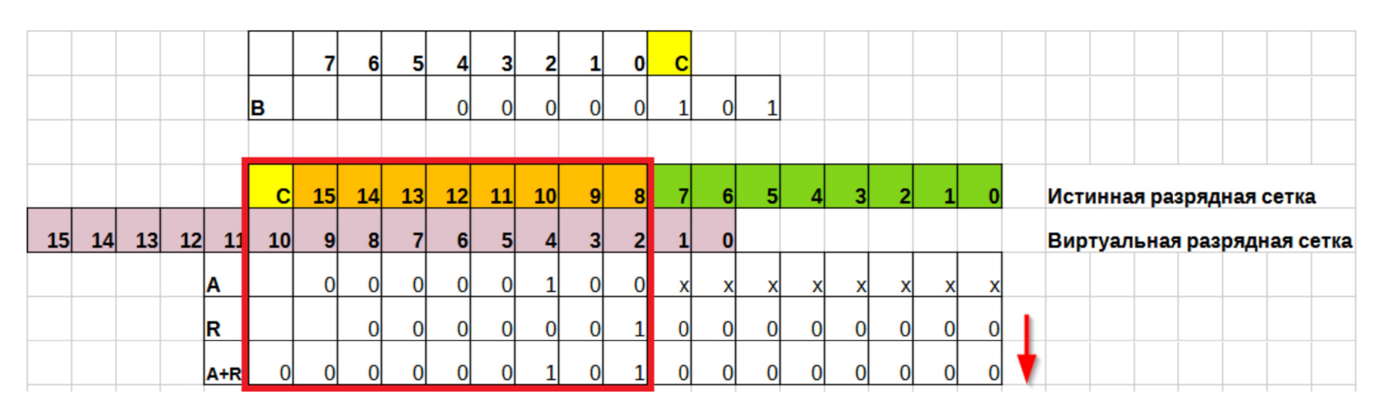


Рисунок 29

Поскольку виртуальная младшая часть A нулевая (выделена крестиками), то истинная младшая часть суммы будет равна младшей части текущего значения R, поэтому достаточно складывать A и R только в пределах старших байт.

Результат сложения записываем обратно в аккумулятор и, как и прежде, сдвигаем аккумулятор вправо на один разряд вместе с виртуальной сеткой, чтобы на следующей итерации получить новое частичное произведение A с весом очередного разряда множителя (если он ненулевой) $A*2^3=4*8=32=2^5$.

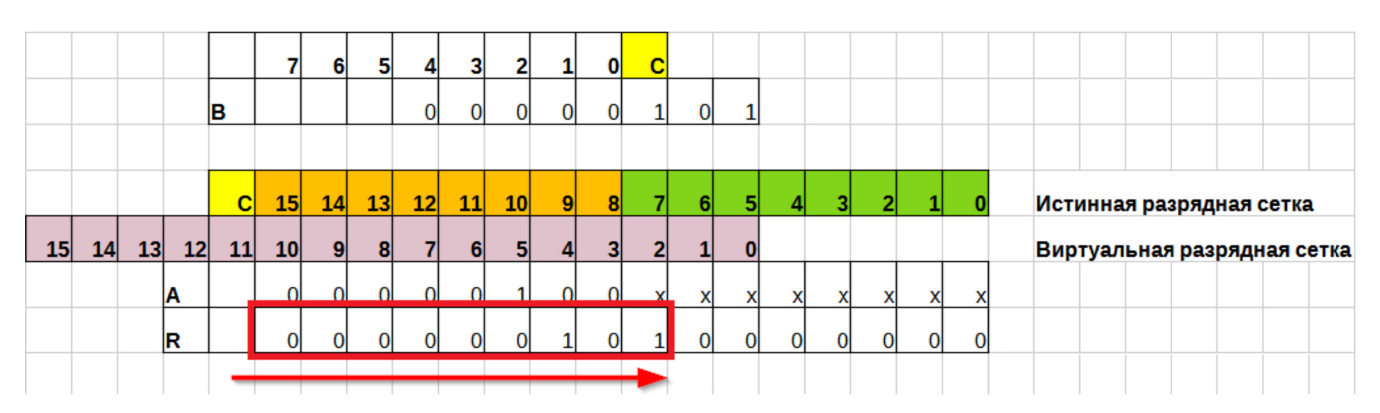


Рисунок 30

Повторяем этот процесс вплоть до старшего бита множителя (MSB – Most Significant Bit). В нашем случае оставшиеся разряды равны нулю, поэтому на всех итерациях мы пропускаем шаг сложения и просто сдвигаем аккумулятор R вправо (вместе с виртуальной разрядной сеткой).

На последней итерации (на последнем разряде множителя) мы получим такой результат:

		7	6	5	4	3	2	1	0	C								
	B									0	0	0	0	0	1	0	1	
	C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Истинная разрядная сетка
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		Виртуальная разрядная сетка
A		0	0	0	0	0	1	0	0	x	x	x	x	x	x	x	x	
R						0	0	0	0	0	0	1	0	1	0	0	0	

Рисунок 31

Множимое A, как и должно быть, относительно сдвинутой виртуальной сетки эквивалентно последнему частичному произведению с весом последнего разряда множителя: $A*2^7=4*128=512=2^9$ (если бы разряд был равен единице, а не нулю).

Важно отметить, что на любом шаге числа A и R могут быть такими, что при сложении могут дать бит переноса влево (выделен желтым).

Еще стоит отметить, что виртуальная разрядная сетка на последней итерации не совпадает с истинной, поэтому в общем случае, даже если после суммирования последнего частичного произведения с аккумулятором R бита переноса нет, мы делаем еще один корректирующий сдвиг аккумулятора R вправо, чтобы одновременно:

1. Совместить виртуальную разрядную сетку с истинной (в пределах двух байт).
2. Вдвинуть бит переноса в старший байт истинного (полного) произведения.

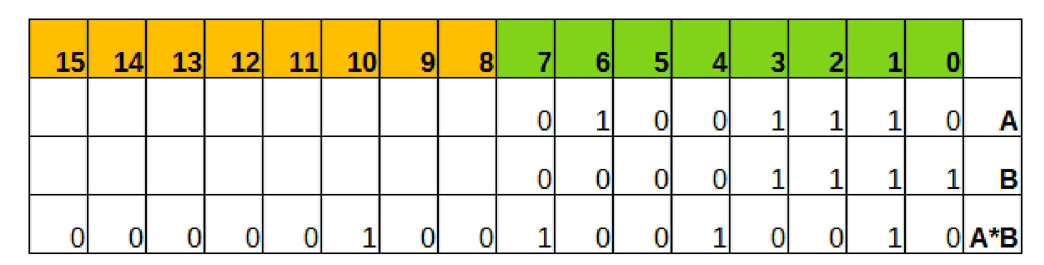
Понятно, что эта схема умножения, по сравнению с базовой, даёт нам возможность выполнять сложение не в двойной сетке, а только в одинарной. То есть, в случае перемножения двух байт, нам достаточно вычислять сумму только в пределах одного (старшего) байта, в отличие от базовой схемы, где после сдвига множимого нам нужно складывать два байта, что, понятное дело, на уровне реализации выльется в дополнительные такты.

Если давать чисто алгебраическую интерпретацию этой схемы, то множимое A в истинной разрядной сетке на первой итерации представляет не свое исходное значение, поскольку находится в старшем байте, а отмасштабированное. Например, на первом шаге, если LSB множителя равен единице, мы прибавляем к аккумулятору не A, а $A*2^7$. Но после прибавления первого частичного произведения следом идёт еще 7 сдвигов: то есть, первое частичное произведение, входящее в полную сумму частичных произведений, в конце умножения, после всех сдвигов аккумулятора вправо, будет фактически поделено на 2^7, т. е., в составе полной суммы будет представлять как раз истинное первое частичное произведение. То же самое справедливо и для всех оставшихся частичных произведений.

Деление целых чисел

Возьмем два числа в пределах байта: $A=78$, $B=15$ и рассмотрим их произведение $C=A*B=78*15=1170$, которое, как мы уже знаем, умещается в двух байтах.

Затем рассмотрим частное от деления: $C/B=A=1170/15=78$. Делимое в худшем случае занимает два байта, но частное вмещается в байт.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
								0	1	0	0	1	1	1	0	**A**
								0	0	0	0	1	1	1	1	**B**
0	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	**A*B**

Рисунок 32

А теперь возьмем $C=4530$ как делимое и $B=15$ — как делитель. Делимое по-прежнему вмещается в два байта, делитель — в байт, а вот частное, равное 302, уже не вмещается в байт.

Обычно, когда мы вычисляем частное, разрядная сетка результата ограничена (вообще, при рассмотрении целочисленного деления, делитель и частное предполагаются в разрядной сетке одинарной длины, а делимое — в двойной). В наших примерах частное ограничено одним байтом. Поэтому, прежде чем выполнять деление, необходимо убедиться, что оно в принципе возможно, то есть, что частное уместится в пределах заданной сетки.

Допустим, что частное ограничено разрядной сеткой длины n, тогда деление возможно только в том, случае, если частное меньше 2^n, а для этого делимое A должно быть меньше делителя, умноженного на 2^n, а это возможно только если разность $A-B*2^n$ отрицательна:

$$A/B<2^n \Rightarrow A<B*2^n \Rightarrow A-B*2^n<0. \qquad (9)$$

Вспомним, что умножение на степень двойки — это сдвиг влево на показатель степени. Тогда, если количество разрядов в одинарной сетке $n=8$, то $B*2^n$ можно получить как сдвиг множителя из младшего байта в старший байт.

То есть, перед началом деления (каким бы образом мы его ни выполняли) мы должны сдвинуть B в старший байт и отнять это значение от делимого A. Если результат вычитания отрицательный, то это значит, что делитель не входит 2^n раз в делимое, то есть, частное меньше 2^n, а значит умещается в пределах байта и задача последующего деления — выяснить это значение.

Базовая схема с восстановлением остатка

Разделим число $A=15$ на число $B=4$ нацело, т. е. с остатком: $A/B=15/4=3$, при этом в остатке оказывается тоже 3.

Действительно, $A=Q*B+R$, где Q – неполное частное, B – делитель, A – делимое, R – остаток: $A=15=3*4+3=15$.

То есть, неполное частное Q от деления A/B равно 3 и остаток R равен 3.

Теперь рассмотрим двоичное представление числа Q в виде суммы степеней двойки: $Q=1*2^1+1*2^0$. То есть, в двоичном виде число Q содержит две единицы: 0b00000011.

Рассмотрим теперь произведение делителя B и неполного частного Q (в разложении по весовым коэффициентам единичных разрядов). Оно, понятное дело, в общем случае будет меньше либо равно делимому A.

$$B*Q=4*(1*2^1+1*2^0)=4*1*2^1+4*1*2^0=4*2+4*1=8+4=12. \qquad (10)$$

Для начала проверим возможность деления способом, описанным в самом начале этой главы.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
$B*2^8$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
$[B*2^8]_{доп}$	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
$A-B*2^8$	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1

Рисунок 33

Разность отрицательная, значит частное A/B меньше $2^8=256$ и помещается в байт. При этом в предельном случае частное может быть равно $2^8-1=255$ и содержать все единицы.

Хотя в этом примере мы уже знаем истинное значение частного, тем не менее, предположим, что нам о нем ничего не известно, тогда мы знаем только, что его значение лежит в $[0,255]$.

Начнем с максимального значения, т. е., предположим, что истинное частное равно 255: $A=B*Q+R$, $15=4*255+R$.

Если это действительно так, то произведение делителя B с частным Q (при делении с остатком частное может быть как полным, так и неполным) даст значение меньше либо равное числу $A=15$, где разность $A-B*Q$, если она ненулевая, будет остатком R.

Разложим предположительное частное $Q=255$ в сумму степеней двойки — это будет сумма весовых коэффициентов всех восьми единичных двоичных разрядов числа 255. Соответственно, произведение $B*Q$ можем представить как сумму частичных произведений числа B с весовыми коэффициентами. А если $Q=255$ — это действительно истинное частное, то, как мы уже сказали выше, $A-B*Q\geq 0$. А в разложении $B*Q$ в сумму частичных произведений это значит, что последовательное вычитание частичных произведений из A также будет давать значения больше либо равные нулю (в случае, если Q – неполное частное, последнее вычитание даст значение меньше B, которое является остатком R).

Реализуем описанный выше процесс. Рассмотрим первый шаг. Весовой коэффициент старшего бита предполагаемого частного Q равен $1*2^7=128$. Умножим B на этот весовой коэффициент и отнимем полученное значение от A.

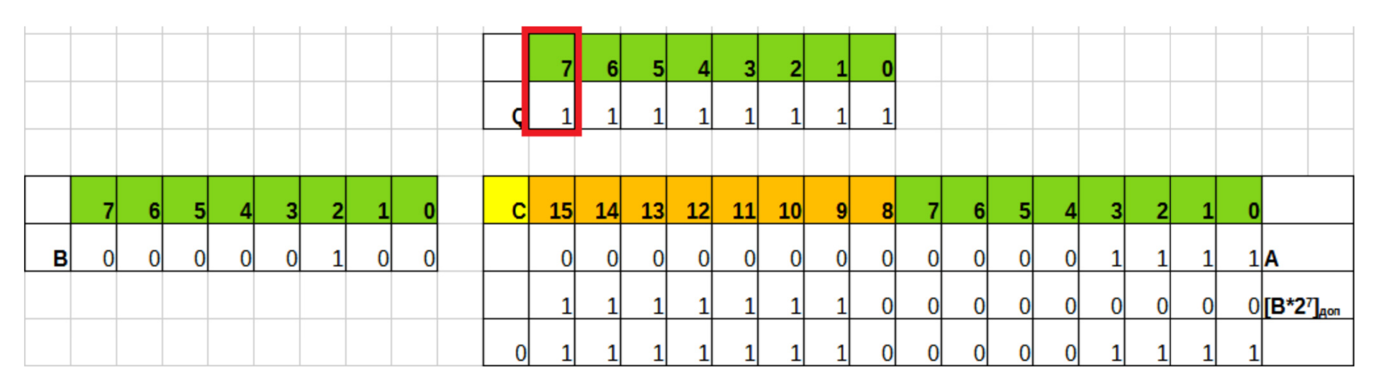


Рисунок 34

Мы получили отрицательное значение. А это значит, что истинное частное меньше $2^7=128$, а значит меньше и 255. То есть, наше предположение оказалось неверным и B не входит в A 255 раз, более того, не входит даже 2^7 раз, как мы выяснили. А это значит, что в двоичном представлении истинного частного в пределах байта старший бит равен нулю (выделен красным на Рисунке 34).

Но какие еще остаются варианты? Возможно, что истинное частное равно $128-1=127$ (все единицы кроме MSB). И снова раскладываем гипотетическое частное в сумму весов и рассматриваем произведение B с этими весами. Как и прежде, если наша гипотеза верна, то разность $A-B*Q_2$ (где Q_2 — очередное гипотетическое частное) даст неотрицательное значение, а значит и последовательная разность с частичными произведениями также будет давать неотрицательные значения.

Но прежде чем мы продолжим проверку, нам нужно вернуться к исходному значению делимого (или, в общем случае, к последнему неотрицательному остатку). Поскольку текущее значение остатка $A'=A-B*2^7$, то для того, чтобы вернуться к исходному значению делимого, нужно прибавить величину, которую мы отняли, т. е. $A=A'+B*2^7$. Это и есть восстановление остатка.

После этого умножаем B на весовой коэффициент старшего единичного разряда нового гипотетического частного Q, равный $1*2^6=64$ и отнимаем от восстановленного делимого A.

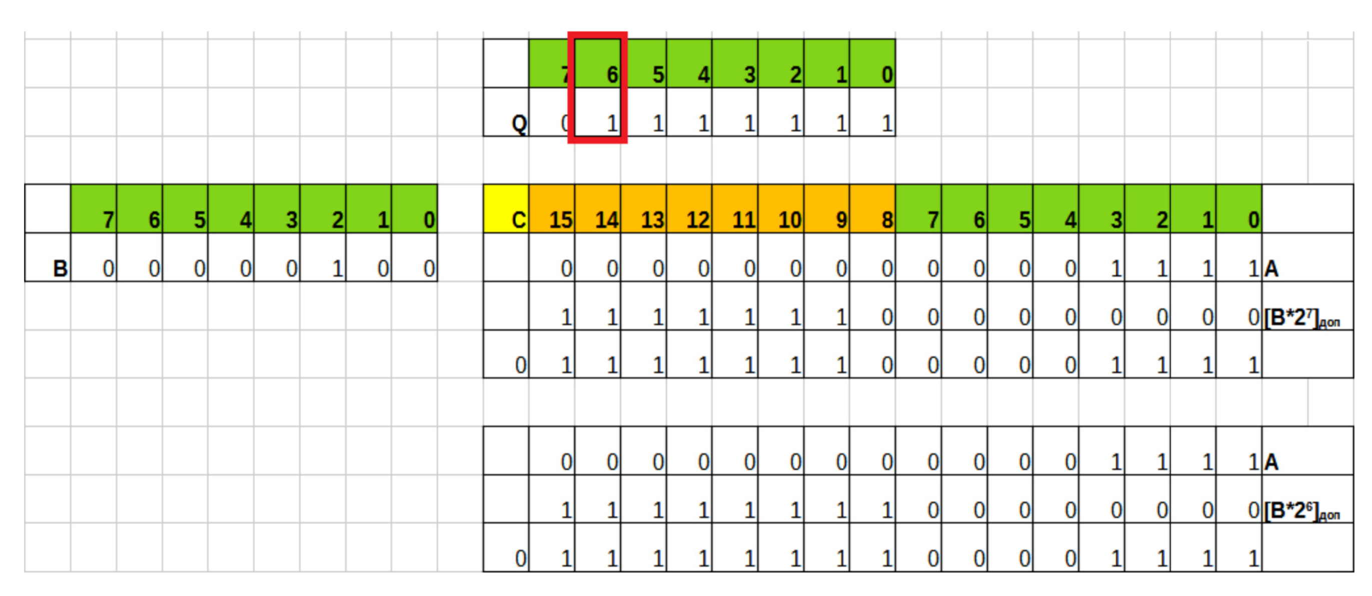
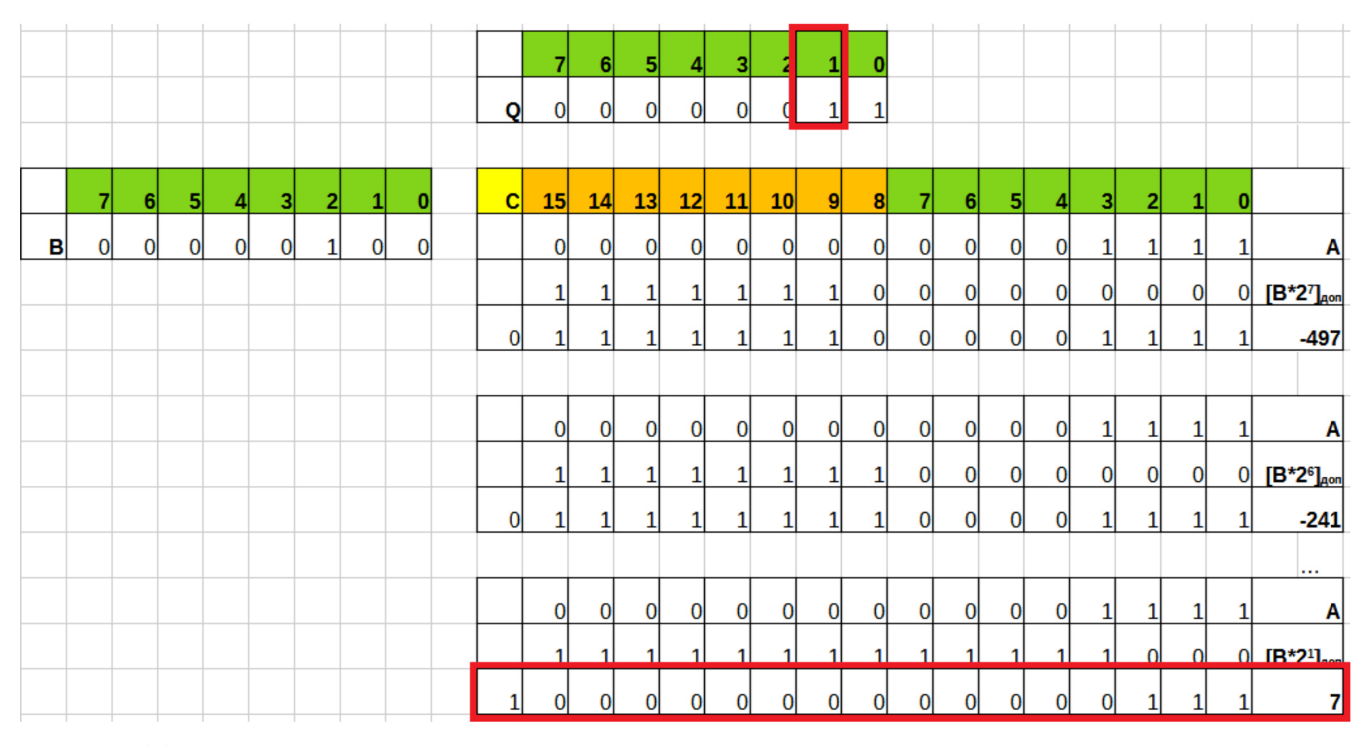


Рисунок 35

И снова разность меньше нуля, то есть, наша новая гипотеза о частном снова оказалась неверной. Это значит, что истинное частное не только меньше $2^7-1=127$, но даже меньше $2^6=64$. Поэтому устанавливаем разряд частного Q в позиции 6 в ноль. А следующая гипотеза — истинное частное Q лежит в $[0,63]$. И она тоже окажется неверной и мы выясним таким образом, что истинное частное меньше $2^5=32$ и разряд истинного частного Q с весом 2^5 равен нулю. Для разрядов с индексами $5,4,3,2$ история повторится и мы в итоге выясним, что истинное частное меньше не только семи (0b00000111), но и меньше $2^2=4$ (0b00000100).

Тогда, если истинное частное меньше четырех, это означает, что оно в предельном случае равно 3. А это снова означает, что раскладывая 3 в сумму весов двоичных разрядов ($1*2^1+1*2^0$) и последовательно вычитая веса из делимого A мы в итоге должны получать неотрицательные значения вплоть до последнего (самого младшего) двоичного разряда.

Умножим делитель $B = 4$ на вес 2^1 и получим 8, отнимем $15 - 8 = 7$. Наша гипотеза частично подтвердилась, истинное частное, как минимум, не меньше двойки.



	7	6	5	4	3	2	1	0
Q	0	0	0	0	0	0	1	1

	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	0

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	$[B*2^7]_{доп}$
0	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	-497
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	$[B*2^6]_{доп}$
0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	-241
																	…
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	$[B*2^1]_{доп}$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7

Рисунок 36

Оставляем разряд частного с весом $1*2^1 = 2$ установленным в единицу, ведь делитель входит в делимое как минимум два раза.

Осталось проверить последний разряд частного с весом $1*2^0=1$.

Наша последняя гипотеза была в том, что делитель входит в делимое $2+1$ раз. В первой части гипотезы мы убедились, получив неотрицательный остаток равный 7. Видно, что и вторая часть тоже справедлива, но всё-равно рассмотрим процесс до конца и убедимся в этом не «выходя» из двоичной системы. Возьмем делитель $B=4$ один раз и отнимем от нового остатка равного 7. Мы получили финальный остаток равный 3.

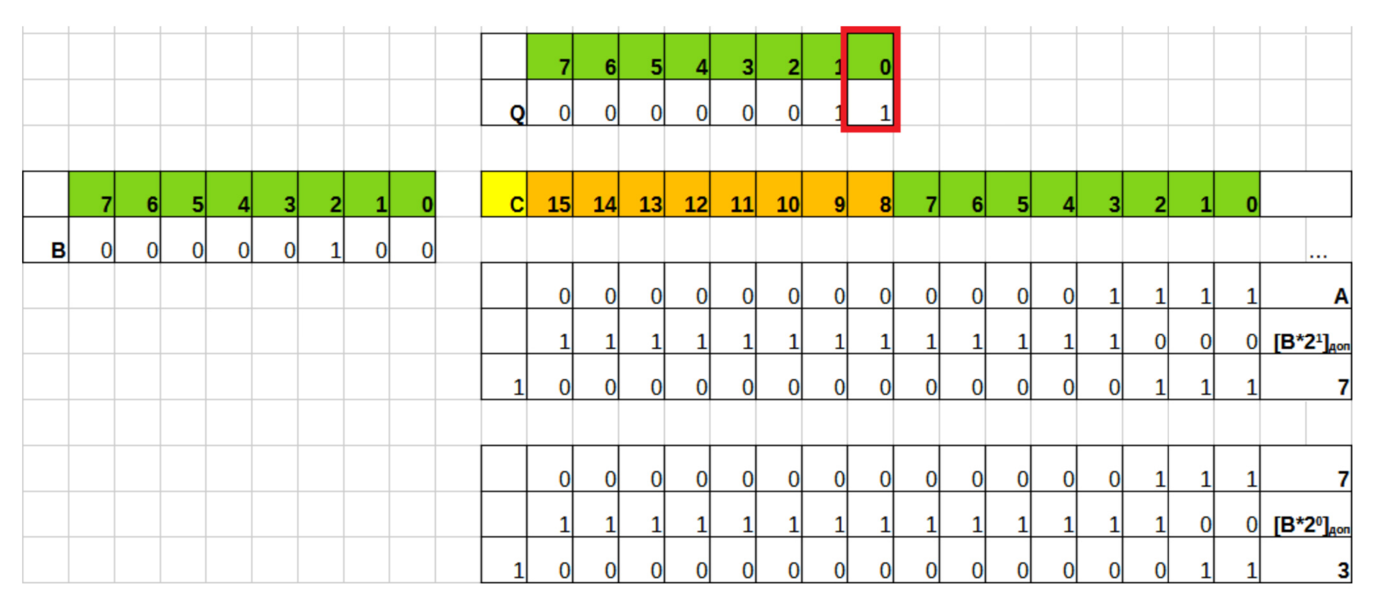


Рисунок 37

То есть, истинное частное от деления равно 3 и остаток равен 3.

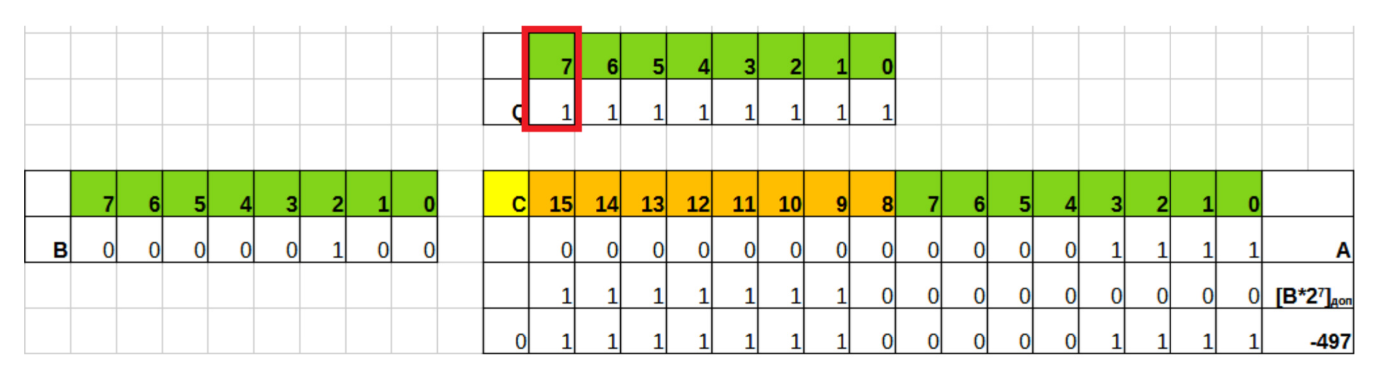
Действительно, $Q*B+R=3*4+3=12+3=15=A$.

Общая идея реализации двоичного деления заключается в том, что, во-первых, мы проверяем возможность деления для заданной разрядной сетки частного (например 8 бит — байт). Убедившись, что деление возможно и не приведёт к переполнению сетки результата, мы, по сути, выяснили, что истинное частное меньше $2^8=256$. Далее мы полагаем, что истинное частное равно максимальному значению 255 (0b11111111), а если это так, то умножая делитель на вес каждого разряда гипотетического частного Q и последовательно вычитая из A мы должны получать неотрицательные остатки (на каждом шаге, включая последний). Но если наша гипотеза о частном не верна и истинное частное меньше 255, то, очевидно, некоторые двоичные разряды истинного частного будут нулевыми и, умножая делитель B на веса этих нулевых двоичных разрядов и вычитая полученное произведение из A (или остатка, если перед этим уже были вычитания) мы будем получать отрицательный результат. После получения отрицательного остатка для данного двоичного разряда, мы устанавливаем разряд в ноль и восстанавливаем остаток до последнего положительного и повторяем процесс для оставшихся разрядов частного.

Базовая схема без восстановления остатка

Продолжим рассматривать те же числа, что и в предыдущем разделе: $A=15$, $B=4$, $C=A/B=3$, $R=3$. Пропустим проверку возможности деления — здесь никаких изменений.

Как и прежде, допустим, что истинное частное равно максимальному значению в пределах байта, т. е. 255. Начнем проверку этой гипотезы с самого старшего разряда предполагаемого частного. Умножим делитель на вес этого разряда: $4*1*2^7=4*128=512$ и отнимем это значение от делимого $A=15$.



	7	6	5	4	3	2	1	0
Q	1	1	1	1	1	1	1	1

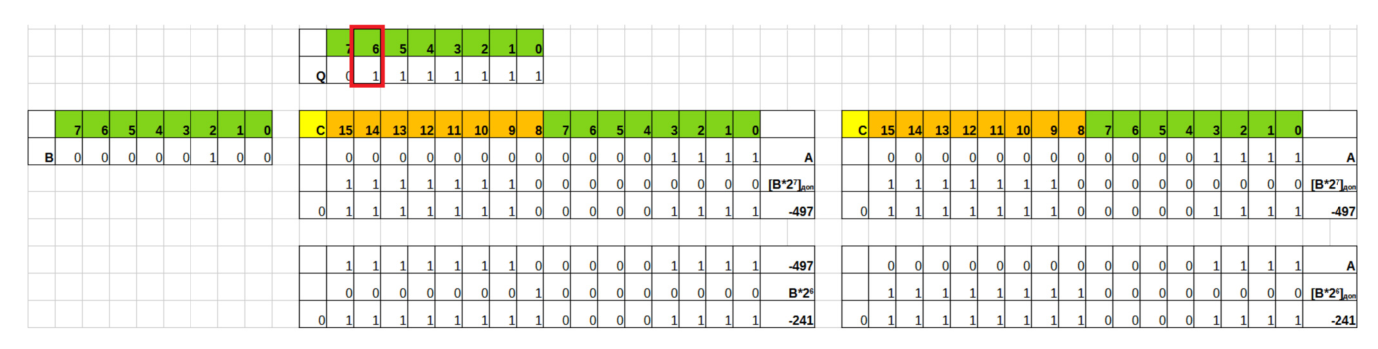
	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	0

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	$[B*2^7]_{доп}$
0	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	-497

Рисунок 38

Остаток отрицательный, поскольку делитель не входит $2^7=128$ раз в делимое, а значит старший разряд истинного частного равен нулю. В схеме с восстановлением остатка, чтобы продолжить процесс деления, мы прибавляли только что вычтенное значение $B*2^7$ к полученному отрицательному остатку, чтобы вернуться к последнему положительному (в данном случае, на первом шаге — к исходному делимому A).

А теперь попробуем оставить отрицательный остаток как есть и перейти к проверке следующего разряда частного с весом 2^6.



	7	6	5	4	3	2	1	0
Q	0	1	1	1	1	1	1	1

	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	0

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	$[B*2^7]_{доп}$
0	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	-497
	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	-497
	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	$B*2^6$
0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	-241

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	$[B*2^7]_{доп}$
0	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	-497
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	$[B*2^6]_{доп}$
0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	-241

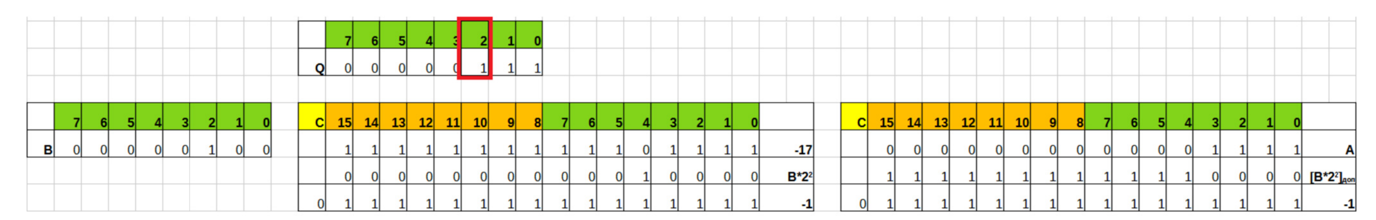
Рисунок 39

Справа изображена прежняя схема, где мы предварительно восстанавливаем последний положительный остаток и только потом выполняем вычитание (само восстановление остатка на схеме не показано для экономии места).

Слева мы пропускаем восстановление остатка и выполняем вместо вычитания прибавление величины $B*2^6=256$, получая при этом то же самое истинное значение нового остатка, что и по прежней схеме.

Новый остаток также отрицательный, поэтому мы повторяем сложение, но уже с величиной $B*2^5$.

Пропустим последующие вычисления и сразу перейдем к шагу, на котором получим последний отрицательный остаток, но после которого должны получить неотрицательный, т. е. — к проверке разряда частного с весом $2^2=4$.



	7	6	5	4	3	2	1	0
Q	0	0	0	0	0	1	1	1

	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	0

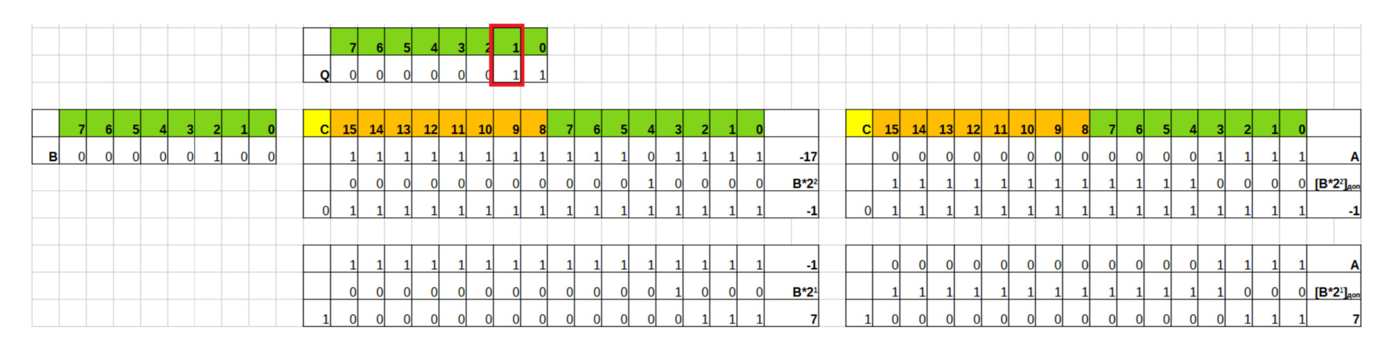
C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	-17
	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	B*2^2
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	$[B*2^2]_{доп}$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1

Рисунок 40

На разряде с весом 2^2 мы снова получили отрицательный остаток (-17 на схеме слева – это остаток с предыдущего шага, на котором мы проверяли разряд частного с весом 2^3: $15-4*2^3=-17$.

Наконец, перейдем к проверке разряда с весом 2^1, и также, вместо восстановления остатка мы сразу прибавим к текущему отрицательному остатку величину $B*2^1$.



	7	6	5	4	3	2	1	0
Q	0	0	0	0	0	0	1	1

	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	0

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	-17
	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	B*2^2
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	B*2^1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	$[B*2^2]_{доп}$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	A
	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	$[B*2^1]_{доп}$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	7

Рисунок 41

И снова мы получили истинный остаток. А поскольку текущий остаток уже положительный, то на следующей (последней) итерации схема ничем не отличается от схемы с восстановлением.

Такая схема деления так и называется — деление без восстановления остатка. Покажем теперь её справедливость более формально.

Обозначим за n длину разрядной сетки частного и делителя (одинарная сетка). А за k обозначим номер шага в алгоритме деления, где k лежит в $[0, n-1]$.

Теперь обозначим текущий остаток, полученный на предыдущем шаге через R_k.

Делитель B, умноженный на вес текущего разряда частного, обозначим как $B'_k = B*2^{(n-1)-k}$

Для $n=8$ на первом шаге $k=0$ это даёт нам $B'_0 = B*2^{(8-1)-0} = B*2^7$, что согласуется с рассмотренной нами схемой.

Теперь рассмотрим разность для произвольного шага в случае, когда она отрицательная:

$$R'_k = R_k - B'_k, R'_k < 0 \quad (11)$$

Мы знаем, что по схеме с восстановлением остатка, для вычисления нового остатка нужно сначала восстановить последний положительный и отнять делитель домноженный на вес следующего разряда частного:

$$R'_{k+1} = (R'_k + B'_k) - B'_{k+1}, B'_{k+1} = B*2^{(n-1)-(k+1)} \quad (12)$$

Раскроем B'_{k+1}:

$R'_{k+1} = R'_k + [B'_k - B*2^{(n-1)-k-1}] = R'_k + [B'_k - B*2^{(n-1)-k}*2^{-1}]$, где $B*2^{(n-1)-k} = B'_k$.

Тогда можно записать:

$R'_{k+1} = R'_k + [B'_k - B'_k*2^{-1}] = R'_k + [B'_k*(1-2^{-1})] = R'_k + [B'_k*2^{-1}]$, где

$B'_k*2^{-1} = B*2^{(n-1)-k}*2^{-1} = B*2^{(n-1)-k-1} = B*2^{(n-1)-(k+1)}$, но это и есть B'_{k+1}.

То есть, действительно, остаток на следующем шаге можно выразить как $R'_{k+1} = R'_k + B'_{k+1}$, поэтому для вычисления нового остатка нет необходимости восстанавливать последний положительный.

Схема с неподвижным делителем с восстановлением остатка

Возьмем два числа: $A=2^{15}=32768$, $B=255$. Но предварительно сделаем несколько модификаций в базовой схеме деления:

1. Расширим разрядную сетку на один разряд влево, добавив т. н. guard-бит.
2. Свяжем с делимым (остатком) подвижную виртуальную сетку.
3. По аналогии со схемой умножения с неподвижным множимым отведем под делитель только один байт.
4. Делитель сдвигать вправо не будем, вместо этого будем сдвигать делимое (остаток) влево.

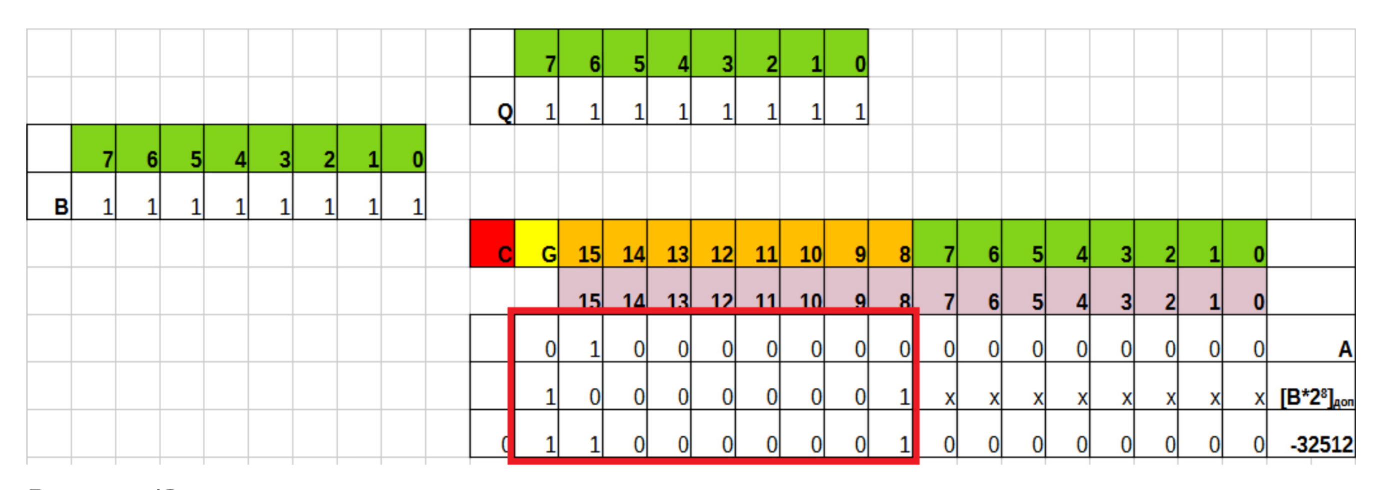


Рисунок 42

Мы видим, что, во-первых, деление возможно, а во-вторых — как и при умножении по схеме не подвижным множителем, нам достаточно теперь оперировать только в пределах старшего байта, поскольку виртуальная младшая часть неподвижного делителя всегда будет нулевой (выделена на рисунке крестиками).

Это был шаг проверки возможности деления. Далее мы, как и в базовой схеме, проверяем старший разряд предполагаемого частного, но делитель при этом оставляем неподвижным, вместо этого сдвигаем делимое влево на один разряд вместе с виртуальной разрядной сеткой.



Рисунок 43

Видим, что делимое не изменило своего значения относительно виртуальной сетки, а вот делитель стал эквивалентен частичному произведению с весом старшего разряда предполагаемого частного. Еще видно причину, почему мы расширили сетку на один разряд влево введя guard-бит: чтобы не потерять значащий разряд делимого (остатка) при сдвиге влево. Ну и снова, поскольку виртуальная младшая часть неподвижного делителя нулевая, истинная младшая часть нового остатка всегда будет равна истинной младшей части делимого (предыдущего остатка), поэтому достаточно выполнять действия только в пределах старшей части.

NOTE: Здесь возникает закономерный вопрос о необходимости и достаточности расширения сетки влево только на один разряд, т. е. не возникнет ли ситуация, когда на следующем сдвиге значащий бит из guard-разряда будет потерян?

Для начала несколько слов о необходимости расширения. Без расширения сетки может возникнуть ситуация переполнения доп. кода отрицательного значения частичного произведения, когда единицы доп. кода окажутся полностью за пределами доступной сетки. При этом значение делимого/остатка может быть меньше частичного произведения, тогда новый остаток будет отрицательным и при этом переполнение доп. кода может остаться (например, если делимое равно $5*2^8$, а делитель равен 255). Хотя на первый взгляд это проблемная ситуация, мы, тем не менее, все еще можем реализовать деление корректно, определяя знак нового остатка не по старшему биту, который может быть равен нулю при отрицательном значении, а по наличию/отсутствию бита переноса. Наконец, рассмотрим граничный случай, когда после сдвига остатка влево мы теряем значащий бит, при этом будем считать, что доп. код отрицательного частичного произведения дает переполнение, то есть, за пределами доступной сетки слева оказывается единица сдвинутого остатка и все единицы доп. кода чатсичного произведения. Очевидно, что независимо от значений операндов, новый истинный остаток при этом всегда будет положительным. При этом значения остатка и частичного произведения могут быть такими, что после сдвига, потери значащего бита и наивного сложения, мы не получим бит переноса, что в обычной ситуации рассматривается нами как признак отрицательного остатка. Но мы помним, что за пределами сетки не только единицы доп. кода, но и значащая единица сдвинутого остатка, что дает зануление виртуальной сетки слева и т.о. дает положительное значение. Таким образом, мы видим, что в случае сдвига положительного остатка, можно обойтись без расширения сетки, но в этом случае реализация может оказаться более сложной из-за необходимости проверки специальных случаев. Поэтому мы выбираем более прямолинейный путь — расширить сетку. После такого расширения доп. код частичного произведения никогда не даст переполнения, и значение нового остатка, если оно отрицательное, также не даст переполнения доп. кода (это мы покажем ниже, после обоснования достаточности расширения).

Теперь покажем, что расширения на один разряд достаточно.

Для начала рассмотрим возможные ситуации:

1. Старшие биты не совмещены. Понятно, что в этом случае всегда $A<B$ и следующий сдвиг влево безопасен, поскольку сдвиг значащего бита A происходит не из guard-разряда.

2. Старшие биты совмещены. Здесь возможно два случая:

    1. $A<B$. Новый остаток отрицательный. В этом случае следующий сдвиг влево также безопасен, т. к. сдвиг значащего бита A происходит как раз в guard-разряд, а не из него.

    2. $A \geq B$. В этом случае остаток неотрицательный, при этом, поскольку старшие биты совмещены, новый остаток в старшем разряде уже будет содержать ноль и последующий сдвиг нового остатка влево также будет безопасным.

3. Старшие биты не совмещены и $A > B$. Новый остаток будет положительным. Это более сложный случай, поскольку значащий бит A уже находится в guard-разряде и возникает вопрос: будет ли значащий разряд нового остатка также находиться в guard-разряде. Если будет, то это плохо, поскольку последующий его сдвиг уже не будет безопасным и мы потеряем значащий бит.

   В этом случае рассуждать будем так: поскольку значащий бит A находится в guard-разряде, значит на предыдущей итерации был сдвиг влево, но раз был сдвиг влево, значит на предыдущей итерации было $A < B$. Более того, старшие биты были совмещены. От этих двух условий и будем отталкиваться.

   То есть, если старшие биты были совмещены и при этом $A < B$, то мы можем представить числа A и B в виде суммы следующим образом: $A = 2^n + \alpha$ и $B = 2^n + \beta$, где 2^n — это вес совмещенных старших разрядов чисел, а $\alpha < \beta$.

   Оставаясь в таком представлении сдвинем A на один разряд влево: $A' = A * 2 = (2^n + \alpha) * 2 = 2^{n+1} + 2\alpha$. А теперь отнимем B: $A' - B = (2^{n+1} + 2\alpha) - (2^n + \beta)$.

   Напомним, что наша задача в том, чтобы убедиться, будет ли у нового остатка нулевой guard-разряд, то есть, будет ли $A' - B$ меньше 2^{n+1}.

   $A' - B = 2^{n+1} + 2\alpha - 2^n - \beta = (2^{n+1} - 2^n) + (2\alpha - \beta) = 2^n + (2\alpha - \beta)$.

   Обнадеживает, осталось только показать, что $(2\alpha - \beta) < 2^n$, т. е. что $2^n + (2\alpha - \beta)$ не даст бит переноса. Поскольку $\alpha < \beta$, то мы можем выразить β так: $\beta = \alpha + \Delta$. Тогда $2\alpha - \beta = 2\alpha - (\alpha + \Delta) = 2\alpha - \alpha - \Delta = \alpha - \Delta$. А поскольку $\alpha < \beta < 2^n$, то полученная разность тем более меньше 2^n, а значит интересующая нас величина $A' - B = 2^n + (2\alpha - \beta)$ меньше 2^{n+1} и т.о. содержит ноль в guard-разряде и следующий сдвиг влево безопасен.

В данном случае мы получили положительный новый остаток, значит старший разряд истинного частного равен единице.

NOTE: После расширения разрядной сетки на один разряд влево отрицательное значение частичного произведения, образованного делителем и весом цифры частного, представляется истинным доп. кодом: то есть, отрицательные и положительные значения частичного произведения различимы по наличию единицы в guard-разряде. А поскольку guard-разряд доп. кода отрицательного значения частичного произведения (для простоты можем называть его просто делителем, хотя он и отмасштабирован в общем случае весом текущей цифры частного) всегда ненулевой, то отрицательное значение нового остатка также будет представлено истинным доп. кодом (здесь может возникнуть вопрос: а если мы получим положительный остаток, может ли у него быть ненулевым guard-бит, но на этот вопрос мы уже ответили отрицательно, обосновывая достаточность расширения сетки влево на один разряд для безопасного сдвига влево в случае схемы с восстановлением остатка). Но поскольку мы сдвигаем делимое влево, то возможна ситуация, когда в расширенной сетке guard-разряд делимого оказывается равным единице. В этой ситуации мы всегда интерпретируем делимое как положительное число, то есть, подразумеваем, что в виртуальной сетке слева не единицы, а

нули. Несмотря на то, что визуально сдвинутое влево положительное значение делимого не отличимо от доп. кода (поскольку старший бит равен единице), путаницы не возникает, поскольку подразумеваются нули а не единицы в виртуальной сетке слева. Более того, поскольку делимое сдвинуто, то результат вычитания с делителем всегда даст в этом сценарии положительное значение.

Продолжаем процесс: сдвигаем (в данном случае уже новый остаток) влево вместе с виртуальной сеткой.

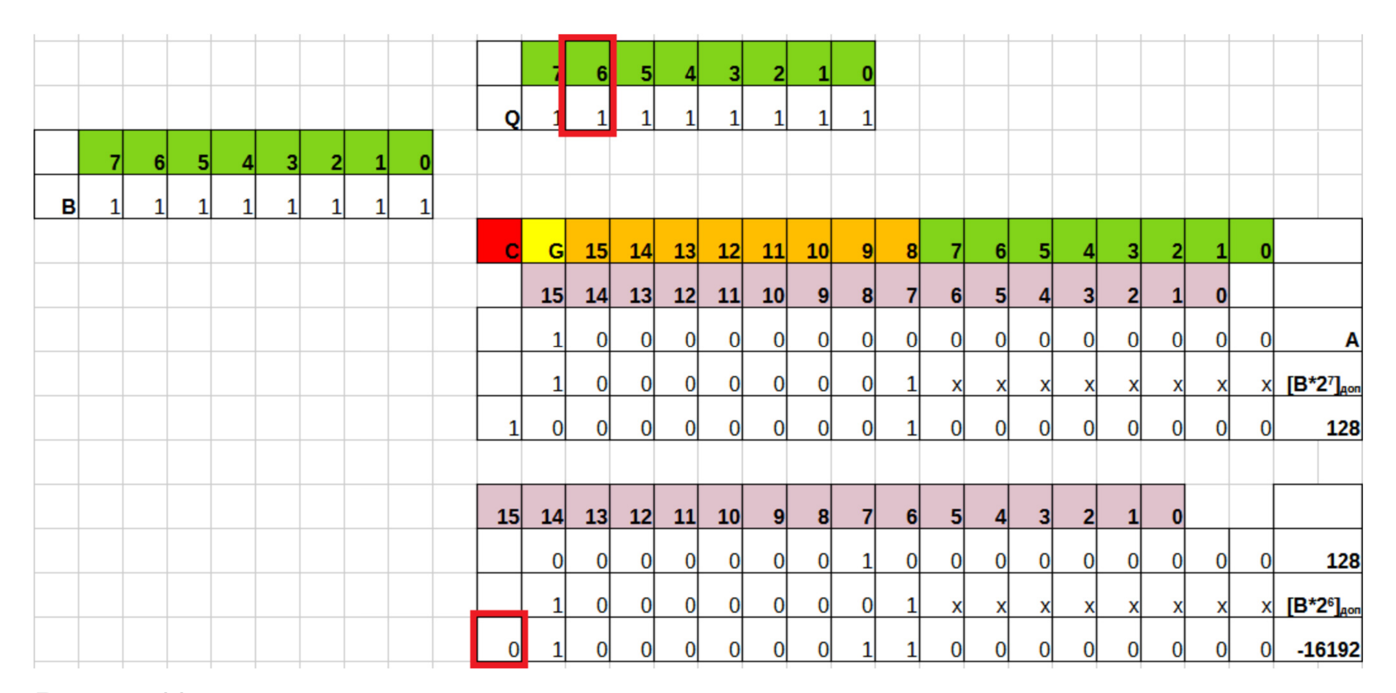


Рисунок 44

И снова относительно виртуальной сетки остаток не меняет своего значения, при этом делитель эквивалентен следующему частичному произведению, а результат (относительно, опять же, виртуальной сетки) эквивалентен истинному результату на данном шаге. Бита переноса нет, поэтому результат отрицательный, а значит следующий бит истинного частного равен нулю.

Восстанавливаем последний положительный остаток равный 128 и повторяем процесс, пока не переберем все биты предполагаемого частного. На самом деле, все последующие итерации будут давать отрицательный остаток, поэтому сразу посмотрим на картину последней итерации.

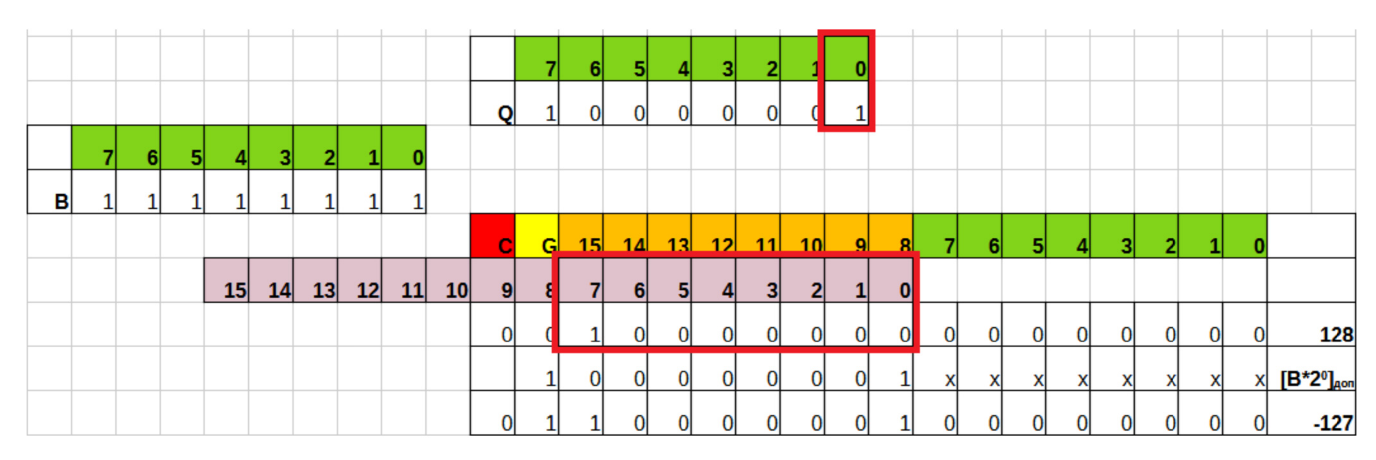


Рисунок 45

Остаток отрицательный — последний бит истинного частного нулевой. Восстанавливаем последний положительный остаток. Видим, что на самом деле, относительно виртуальной сетки, истинный остаток будет содержаться целиком в старшей части результата.

Действительно, $128*255+128=32768$.

Схема с неподвижным делителем без восстановления остатка

Схема с неподвижным делителем с восстановлением остатка в трактовке через виртуальную сетку эквивалентна базовой схеме с восстановлением, поэтому её легко можно расширить до схемы без восстановления остатка (которую мы уже обосновали для базовой), достаточно только в случае получения отрицательного остатка сдвигать его влево, а не восстанавливать последний положительный. Это не изменит значения отрицательного остатка относительно виртуальной сетки, но даст новое частичное произведение делителя с весом очередной цифры частного, которое мы и сложим с отрицательным остатком.

Именно эту, последнюю, вариацию алгоритма деления мы и будем в дальнейшем использовать при реализации деления чисел в формате плавающей точки. Разница будет лишь в том, что операнды будут не целыми числами, а дробными. А как представлять дробные числа в двоичном виде мы рассмотрим в следующем разделе.

NOTE: Мы уже расширили сетку влево на один разряд для схемы с восстановлением остатка и показали достаточность такого расширения для безопасного сдвига положительного остатка влево. При этом мы показали отсутствие необходимости расширения, но выбрали расширение для более простой (и, как далее увидим, более универсальной) реализации. А поскольку отрицательный остаток, который мы также будем сдвигать влево, в схеме без восстановления представлен в доп. коде, нужно показать, что этого расширения, которое мы уже сделали, будет достаточно и для безопасного сдвига доп. кода.

Но сперва скажем пару слов о необходимости расширения в случае сдвига доп. кода (временно забудем о том, что в данной схеме при делении у нас возникает необходимость сдвига как положительного значения так и отрицательного). В схеме с восстановлением остатка мы брали в качестве примера делимое равное $5*2^8$ и делитель 255. Если мы не расширяем сетку, то уже на первой итерации получаем отрицательный остаток с переполнением доп. кода. Поскольку сейчас мы рассматриваем схему без восстановления остатка, то на следующей итерации мы сдвигаем доп. код отрицательного остатка влево, а поскольку уже имеем переполнение доп. кода, то за пределы доступной сетки сдвигается 0. После прибавления неподвижного делителя, мы получаем новый отрицательный остаток, который также дает переполнение доп. кода и при этом в MSB содержит ноль. Более того, он дает бит переноса за пределы доступной сетки, который «падает» в свинутый ранее ноль.

	C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	$5*2^8$
	1	0	0	0	0	0	0	0	1	x	x	x	x	x	x	x	x	-255*28
	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	-64000
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0		-64000*2
		1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	+255*28
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	-62720

Рисунок 46 (числа определены относительно неподвижной сетки)

То есть, во-первых, без расширения сетки мы не можем корректно определять знак остатка по старшему биту (в нашем примере, у нового остатка бит равен нулю, но истинный остаток тем не менее отрицательный), а во-вторых, мы не можем определять знак даже по биту переноса (в нашем примере бит переноса есть, а это признак положительного числа (в схеме с восстановлением мы показали, что когда есть переполнение доп. кода и все единицы за пределами сетки, бит переноса их занулит и тем самым даст положительное значение), но он падает в сдвинутый ноль, тем самым оставляя истинный результат отрицательным. Поэтому в случае сдвига доп. кода влево, имеет смысл расширить сетку влево, получив тем самым более простую реализацию.

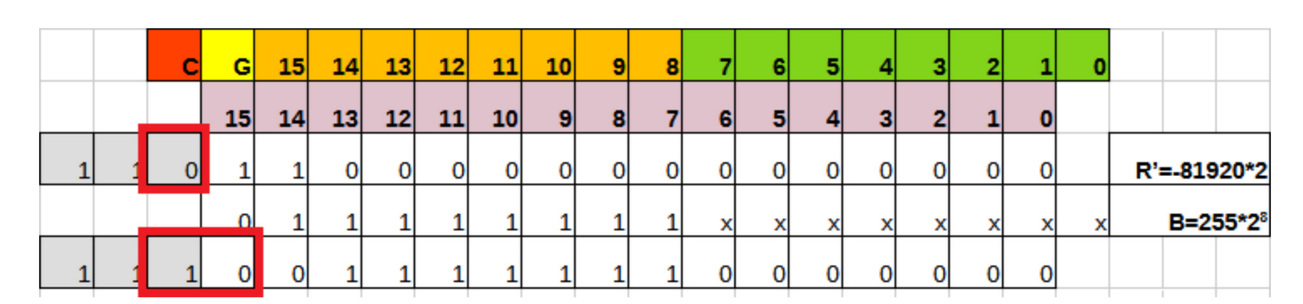
Таким образом, поскольку в схеме без восстановления с неподвижным делителем требуется сдвиг как положительного остатка так и отрицательного, в общем случае проще всего расширить сетку влево, получив возможность просто и корректно определять знак остатка. А достаточность этого расширения для безопасного сдвига доп. кода влево мы покажем ниже.

Для начала определим, когда сдвиг будет безопасным. Для этого рассмотрим следующий пример, в котором под делимое (остаток) отведено два байта, а под делитель и частное — один. При этом значение делителя B возьмем наибольшее в пределах байта — 255. Далее допустим, что на предыдущей итерации мы получили отрицательный остаток $R = -81920$, который даже в расширенной разрядной сетке дает переполнение доп. кода: то есть, единицы доп. кода оказываются за пределами фактический разрядной сетки слева (эта область выделена на рисунке серым цветом).

	C	G	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	R=-81920
		0	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	$B=255*2^8$

Рисунок 47

Напомним, что поскольку новый остаток отрицательный, то по схеме без восстановления остатка, чтобы получить следующий остаток, мы должны прибавить к нему частичное произведение делителя с весом следующей младшей цифры частного. А поскольку мы делим по схеме с неподвижным делителем, то перед этим сдвигаем не делитель, а остаток влево вместе с виртуальной разрядной сеткой (фиолетовая).



		C	G	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0		R'=-81920*2
			0	1	1	1	1	1	1	1	1	x	x	x	x	x	x	x	x	$B=255*2^8$
1	1	1	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0		

Рисунок 48

Видим, что после сдвига остатка влево в виртуальную часть «ушел» ноль. Более того, после сложения с частичным произведением, мы получили отрицательный результат, который также как и предыдущий остаток до сдвига дает переполнение доп. кода. При этом мы не можем однозначно определить отрицательный знак нового остатка не только по старшему биту G, который теперь равен нулю, но даже по отсутствию бита переноса, поскольку в данном случае бит переноса есть, но «падает» он в нулевой разряд слева в виртуальной сетке, поэтому результат остается отрицательным.

Теперь мы можем определить достаточность расширения сетки на один разряд влево в случае сдвига доп. кода при делении по схеме без восстановления остатка с неподвижным делителем. Расширения сетки влево будет достаточно, когда после сдвига текущего остатка влево и

сложения с частичным произведением, новое значение остатка не будет давать переполнение доп. кода в расширенной сетке.

Мы видим, что после сдвига последнего остатка влево может возникнуть переполнение доп. кода. Вопрос в том, после сложения с частичным произведением, останется ли это переполнение доп. кода?

Сразу отметим, что далее все величины будут оцениваться относительно реальной разрядной сетки, а не подвижной виртуальной.

Продолжим рассуждения с того факта, что деление мы начинаем с положительного значения делимого и делителя (напомним, что знак частного вычисляется отдельно, а сам процесс деления выполняется над модулями). То есть, на самой первой итерации мы к положительному значению делимого прибавляем доп. код отрицательного значения частичного произведения делителя с весом цифры частного, которая следует после самой старшей цифры частного (первая итерация — проверка возможности деления). При этом, наименьшее ненулевое значение делимого равно единице, а наибольшее значение делителя в пределах байта равно 255.

То есть, на первой итерации наибольшее по модулю отрицательное значение остатка, которое мы можем получить будет равно $1-255*2^8=-65279$. При этом, поскольку делимое всегда ненулевое (когда делимое равно нулю, на уровне реализации мы обрабатываем этот случай отдельно), то новый остаток по модулю будет меньше первого частичного произведения. При этом, отрицательное значение наибольшего частичного произведения в расширенной сетке никогда не даст переполнения доп. кода, а из этого следует, что отрицательное значение нового остатка тем более не даст переполнения доп. кода.

Далее, поскольку новый остаток отрицательный, то для вычисления очередного остатка мы сдвигаем его влево, получая $-65279*2=-130558$. Это значение дает переполнение доп. кода. Но далее мы прибавляем все то же значение частичного произведения делителя, которое изначально дало отрицательный остаток: $-130558+(255*2^8)=-65278$, а это значение оказывается по модулю меньше чем постоянное частичное произведение делителя равное 65280. То есть, новое значение отрицательного остатка не только оказывается меньше, чем отрицательное значение частичного произведения делителя, но и меньше (по модулю) предыдущего отрицательного остатка. Таким образом, новый остаток не дает переполнения доп. кода и его знак мы можем определить как по G-разряду, так и по наличию/отсутствию бита переноса (именно так мы проверяем на уровне реализации).

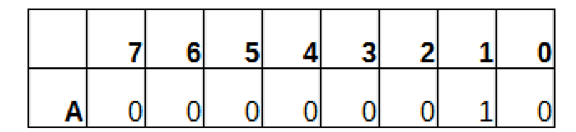
Если мы продолжим рассуждения, то увидим, что новый остаток оказался по модулю не только меньше частичного произведения делителя, но и предыдущего отрицательного остатка, а это значит, что после очередного сдвига влево и прибавления частичного произведения, новое значение отрицательного остатка окажется еще ближе к нулю.

Хотя рассуждения выше нельзя считать строгим обоснованием достаточности расширения сетки влево на один разряд, интуитивно должно быть понятно, что и на последующих итерациях история повторится и что после умножения очередного отрицательного остатка (которые по модулю меньше предыдущего остатка и меньше частичного произведения) на 2 и прибавления

частичного произведения, если новый остаток будет отрицательным, то он никак не даст переполнения доп. кода и поэтому будет содержать ненулевой G-бит, а значит будет безопасным для очередного сдвига влево и сложения с частичным произведением. Постепенно такие сдвиги и прибавления будут давать остаток всё ближе и ближе к нулю слева направо, в конечном итоге давая либо нулевой остаток либо ненулевой положительный, который будет меньше делителя.

Представление дробных чисел

Возьмем число $A=2$ в разрядной сетке размера $n=8$, т. е. в пределах байта.



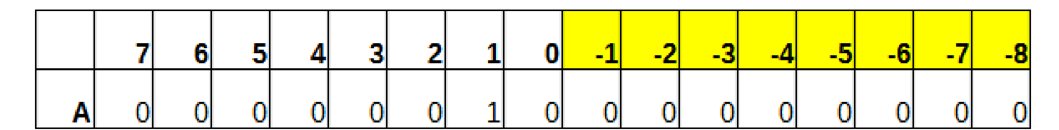
	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	1	0

Рисунок 49

Это представление целого числа 2 в двоичной системе в пределах байта. Напомним, что каждому двоичному разряду соответствует весовой коэффициент, если разряд равен единице. То есть, эта двоичная цепочка однозначно определяет десятичное число $A=2$.

$$A=0*2^7+0*2^6+0*2^5+0*2^4+0*2^3+0*2^2+1*2^1+0*2^0=1*2^1=2 \quad (13)$$

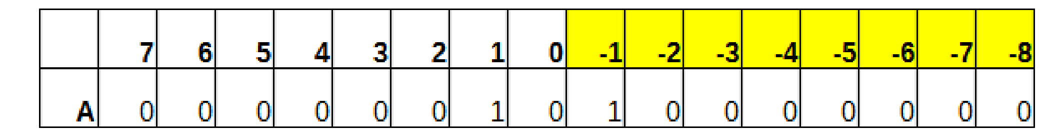
А теперь расширим разрядную сетку вправо еще на 8 разрядов, но пронумеруем их уже отрицательными числами.



	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Рисунок 50

И установим разряд в позиции -1 в единицу.



	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0

Рисунок 51

По уже известному нам принципу свяжем с этим разрядом весовой коэффициент. Но поскольку индексы разрядов отрицательные, то весовой коэффициент будет не положительной степенью двойки, а отрицательной:

$$A=1*2^1+0*2^0+1*2^{-1}=2+1*\frac{1}{2}=2+0.5=2.5. \quad (14)$$

Как видим, весовой коэффициент единицы в позиции -1 по этому принципу равен 0.5 и вместе с весовым коэффициентом единицы в позиции 1 однозначно определяет дробное число 2.5.

Понятно, что все двоичные единицы с отрицательными индексами вправо будут давать дробные числа, т. к. $2^{-k}=\frac{1}{2^k}$, то есть, это всегда будет дробь меньше единицы.

Поэтому мы можем мысленно между индексом 0 и -1 поставить двоичную точку. Также, как десятичная точка отделяет целые десятичные разряды от дробных, так и здесь двоичная точка отделяет целые двоичные разряды от дробных двоичных.

А теперь возьмем число B равное числу A и сложим, как мы это делали в самом начале, когда разбирали представление целых чисел в двоичном виде.

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
B	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0

Рисунок 52

Мы уже знаем, что истинный результат в десятичном виде будет равен $A+B=2.5+2.5=2+0.5+2+0.5=2+2+0.5+0.5=4+1=5$.

Сразу запишем его в двоичном виде под числами A и B:

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
B	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
R	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0

Рисунок 53

Мы видим, что в случае дробных чисел сохраняется та же закономерность при сложении: две двоичные единицы в одинаковой позиции при сложении дают ноль в этой позиции и бит переноса в следующую позицию слева.

Действительно, $0.5+0.5=1$, а это двоичная единица в позиции 0, вес которой равен $1*2^0$. Ну а сложение целочисленных двоичных единиц мы уже подробно разбирали.

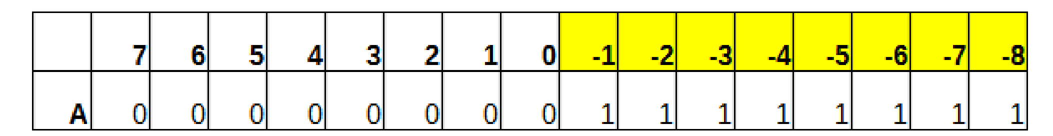
А теперь заполним целую часть всеми единицами.

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

Рисунок 54

Сумма их весовых коэффициентов будет равна:
$A=1*2^7+1*2^6+1*2^5+1*2^4+1*2^3+1*2^2+1*2^1+1*2^0=128+64+32+16+8+4+2+1=255.$

А теперь повторим то же самое для дробной части.



	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Рисунок 55

Сумма весовых коэффициентов будет равна:

$A=1*2^{-1}+1*2^{-2}+1*2^{-3}+1*2^{-4}+1*2^{-5}+1*2^{-6}+1*2^{-7}+1*2^{-8}=$
$0.5+0.25+0.125+0.0625+0.03125+0.015625+0.0078125+0.00390625=0.99609375$

Это максимальное дробное значение меньше единицы, которое мы можем представить в разрядной сетке размера 8.

А теперь просто сложим максимальное целое и максимальное дробное и получим 255.99609375 – это максимальное дробное значение, которое мы можем представить в двух байтах, где старший байт отведен под целую часть, а младший — под дробную.

Вся «магия» только в весовых коэффициентах, то есть, в том, как мы интерпретируем двоичные разряды.

Сложение дробных чисел

Возьмем два числа: $A=1.25$ и $B=0.25$ и сложим их по тому же принципу, по которому мы складываем целые числа. Получится 1.5.

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
A+B	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

Рисунок 56

Как видим, введение дробных чисел принципиально ничего не изменило и сложение выполняется по тем же законам, что и для целых чисел — два единичных бита в одинаковой позиции при сложении дают ноль в сетке результата в этой же позиции и бит переноса в следующую позицию.

Теперь посмотрим на сложение немного иначе. Возьмем те же числа A и B и домножим каждое на 2^8, то есть, сдвинем влево на 8 разрядов. При этом расширим сетку влево еще на 8 разрядов а сетку с отрицательными индексами временно отбросим.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0								
B	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0								

Теперь мы имеем числа: $A'=A*2^8=320$ и $B'=B*2^8=64$. Действительно, если сложить весовые коэффициенты, то мы получим именно эти числа.

Сложим их и получим 384.

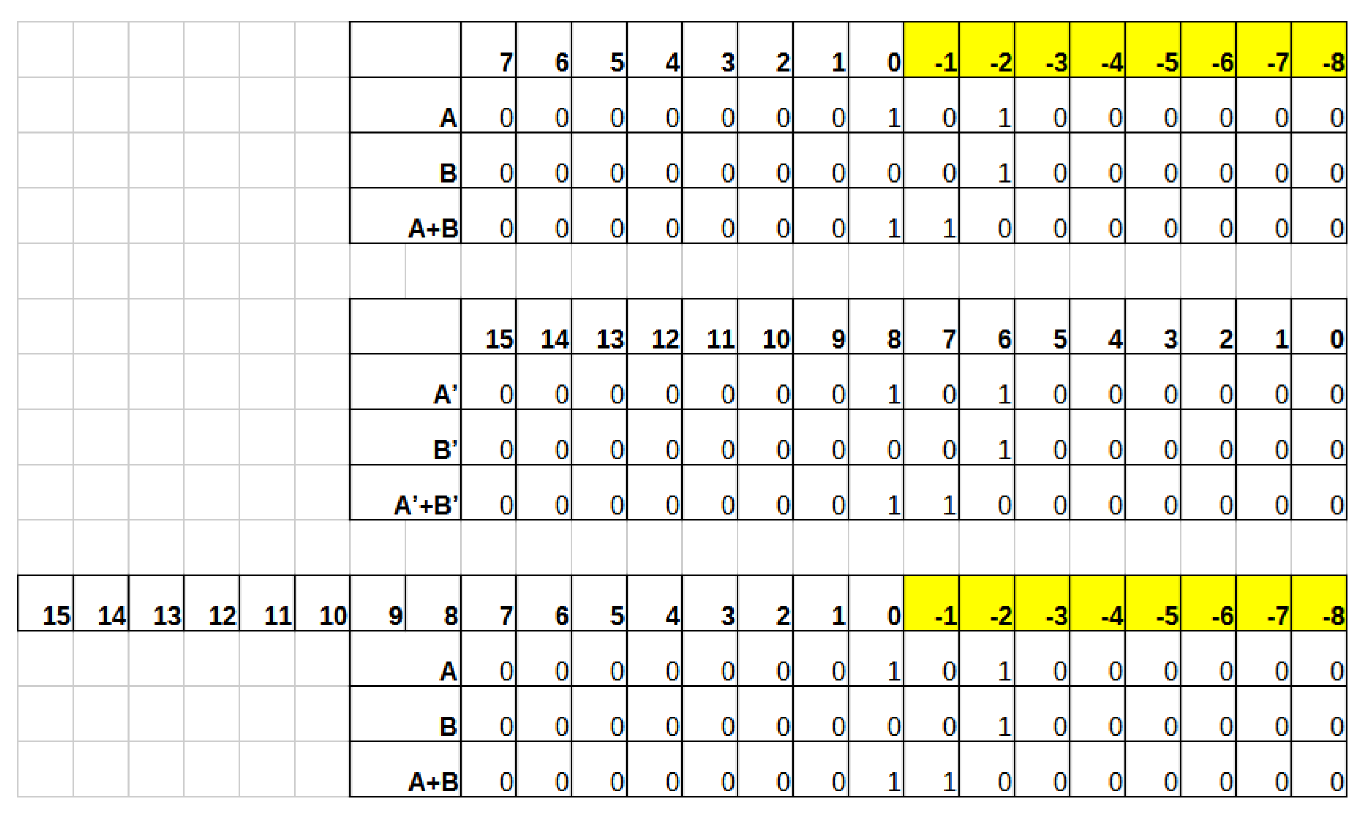
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A’	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
B’	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
A’+B’	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

Рисунок 58

Распишем теперь полученный результат: $A'+B'=A*2^8+B*2^8=(A+B)*2^8$.

То есть, результат, который мы получили, оказался отмасштабирован на тот же коэффициент, что и входные операнды. Таким образом, чтобы вернуться к истинному результату, достаточно разделить полученный результат на 2^8, т. е. сдвинуть вправо на 8 разрядов.

Но для возврата к истинному значению мы можем просто изменить интерпретацию полученного результата и мысленно сдвинуть индексы сетки влево на 8 разрядов, восстановив отброшенную её часть с отрицательными значениями.



	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
A+B	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A’	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
B’	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
A’+B’	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
							A	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
							B	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
							A+B	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

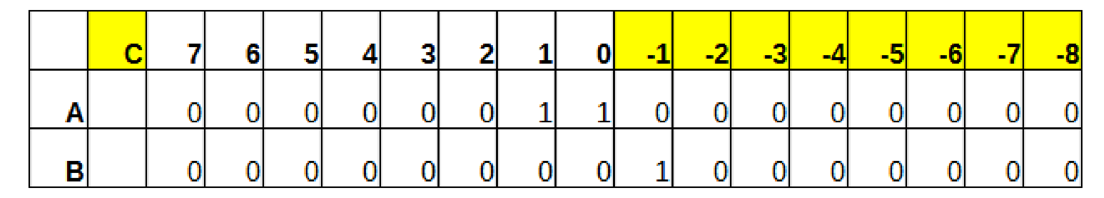
Рисунок 59

Как видим, получается тот же самый результат, что и при сложении исходные дробных операндов.

Таким образом, если при сложении дробных чисел сначала мысленно перейти к целым значениям, домножив исходные дробные числа на соответствующий масштабный коэффициент, то сложение будет эквивалентно целочисленному сложению, которое мы уже подробно разобрали. А чтобы вернуться к истинному результату, достаточно мысленно разделить полученный результат на тот же самый масштабный коэффициент. Но, конечно, можно выполнить сложение и без мысленного перехода, непосредственно оперируя дробными весовыми коэффициентами.

Вычитание дробных чисел

Возьмем числа $A = 3$ и $B = 0.5$ и вычислим их разность $A - B = 3 - 0.5 = 2.5$.



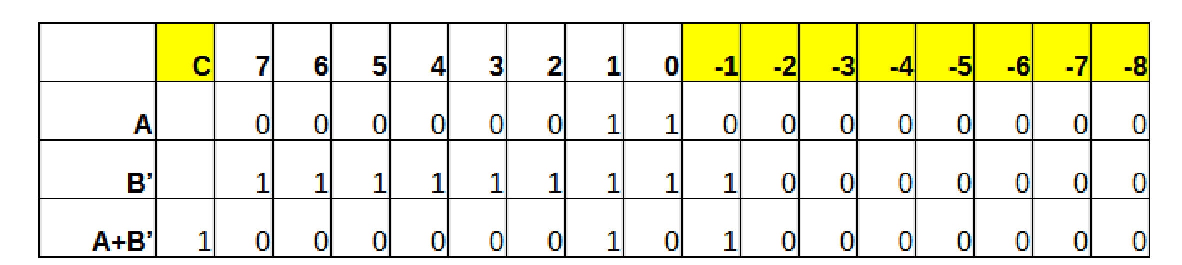
	C	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A		0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
B		0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

Рисунок 60

Напомним, что для представления отрицательного числа -0.5 мы используем доп. код, то есть, дополнение до переполнения текущей разрядной сетки.

Вспомним суть доп. кода. В нашем случае нас интересует такое число в пределах данной сетки, которое при сложении с 0.5 даст переполнение этой сетки и т.о. зануление всех разрядов. Максимальное значение в пределах нашей сетки равно (с учетом дробной части) 255.99609375 и если мы прибавим к нему 2^{-8} (вес наименьшей двоичной единицы) то получим 256 — число, которое уже не представимо в пределах текущей сетки и даёт зануление. То есть, нас интересует число, которое при сложении с $B = 0.5$ даст 256, это и будет доп. код отрицательного числа -0.5.

$$B' = 2^8 - B = 2^8 - 0.5 = 256 - 0.5 = 255.5 . \quad (15)$$



	C	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A		0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
B’		1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
A+B’	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0

Рисунок 61

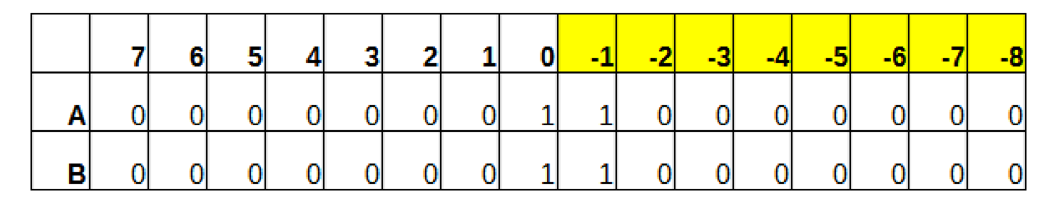
Как видим, принцип вычисления разности в случае дробных двоичных чисел тот же, что и для целых чисел.

Понятно, что как и в случае сложения, мы можем перейти мысленно к целым, отмасштабировав исходные значения, вычислить разность между целыми числами и разделить полученный результат на масштабный коэффициент, вернувшись к истинному дробному результату.

Умножение дробных чисел

Базовая схема

Возьмем два числа: $A=1.5$ и $B=A=1.5$ и вычислим их произведение, которое будет равно 2.25.



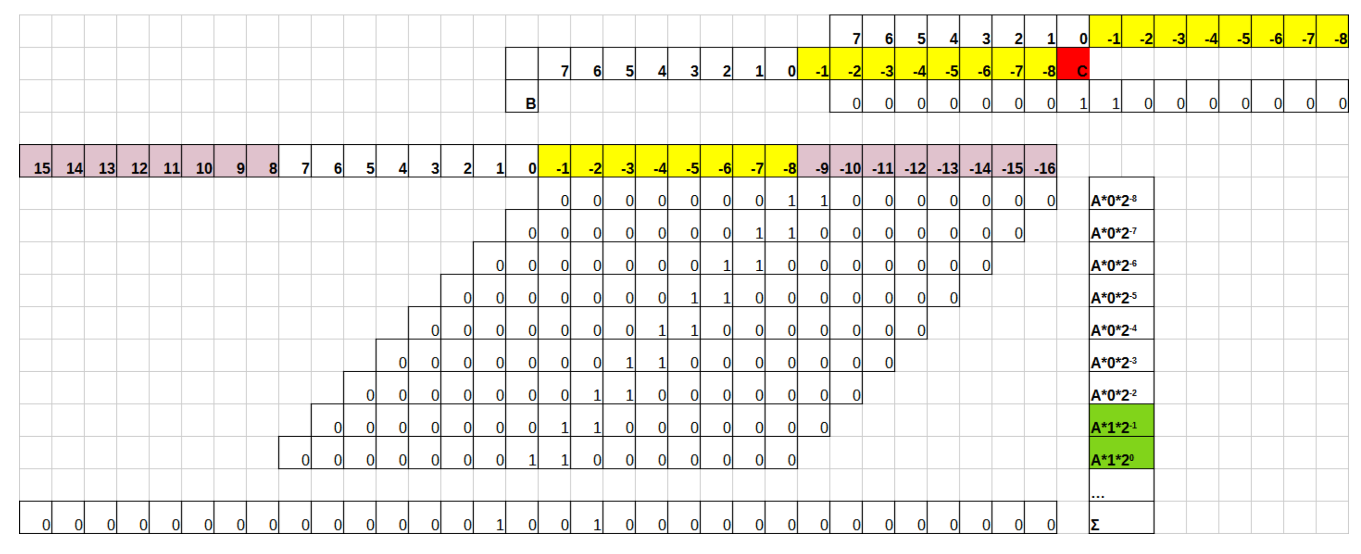
	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0

Рисунок 62

Вычислять будем по базовой схеме: суммируя частичные произведения числа A с весовыми коэффициентами ненулевых двоичных разрядов числа B.

Но перед этим расширим разрядную сетку множимого A влево и вправо на байт, поскольку будем умножать на веса 2^7 и 2^{-8} соответственно (выделено фиолетовым на рисунке). И еще, в отличие от целочисленного умножения, исходное значение множимого A нужно сразу интерпретировать как $A*2^{-8}$, поскольку вес самого младшего разряда множителя B, с которого мы начинаем умножение, равен $1*2^{-8}$, а не $1*2^0=1$, как в случае целого числа.

Для экономии места опустим частичные произведения с весами нулевых целочисленных двоичных разрядов числа B.



																									7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
																7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	C								
															B										0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16									
																0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0		$A*0*2^{-8}$							
															0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0			$A*0*2^{-7}$							
														0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0				$A*0*2^{-6}$							
													0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0					$A*0*2^{-5}$							
												0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0						$A*0*2^{-4}$							
											0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0							$A*0*2^{-3}$							
										0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0								$A*0*2^{-2}$							
									0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0									$A*1*2^{-1}$							
								0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0										$A*1*2^{0}$							
																																	...							
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0		Σ							

Рисунок 63

Мы видим, что умножение дробных чисел принципиально ничем не отличается от умножения целых чисел: логика та же, просто весовые коэффициенты дробной части имеют нецелочисленную интерпретацию.

Но также, как и в случае сложения/вычитания, мы можем предварительно перейти от дробных операндов к целочисленным и выполнить более привычное целочисленное умножение.

$$A' = A * 2^8 = 1.5 * 2^8 = 384,$$
$$B' = B * 2^8 = 1.5 * 2^8 = 384,$$
$$A' * B' = 1.5 * 2^8 * 1.5 * 2^8 = 1.5 * 1.5 * 2^8 * 2^8 = (A * B) * 2^{16} = (A * B) * (2^8)^2.$$

То есть, после предварительного масштабирования операндов умножением на 2^8, произведение оказывается больше истинного в $(2^8)^2$. Таким образом, для возврата к истинному произведению, нужно разделить результат на квадрат масштабного коэффициента, на который был домножен каждый операнд.

Ранее мы уже убедились, что умножение целых чисел можно выполнить как по базовой схеме так и по схеме с неподвижным множимым. А поскольку предварительный переход к целым принципиально ничего не меняет, кроме индексов разрядной сетки, понятно, что умножение по схеме с неподвижным множимым справедливо и для дробных чисел.

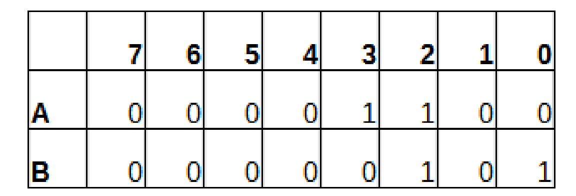
Деление дробных чисел

Поскольку, как мы увидим далее, деление дробных технически ничем не отличается от деления целых, для простоты рассмотрим только базовую схему с восстановлением остатка.

Базовая схема с восстановлением остатка

Деление целых с дробным результатом

Возьмем два целых числа в пределах байта: $A=12$ и $B=5$. Под частное тоже отведем один байт.



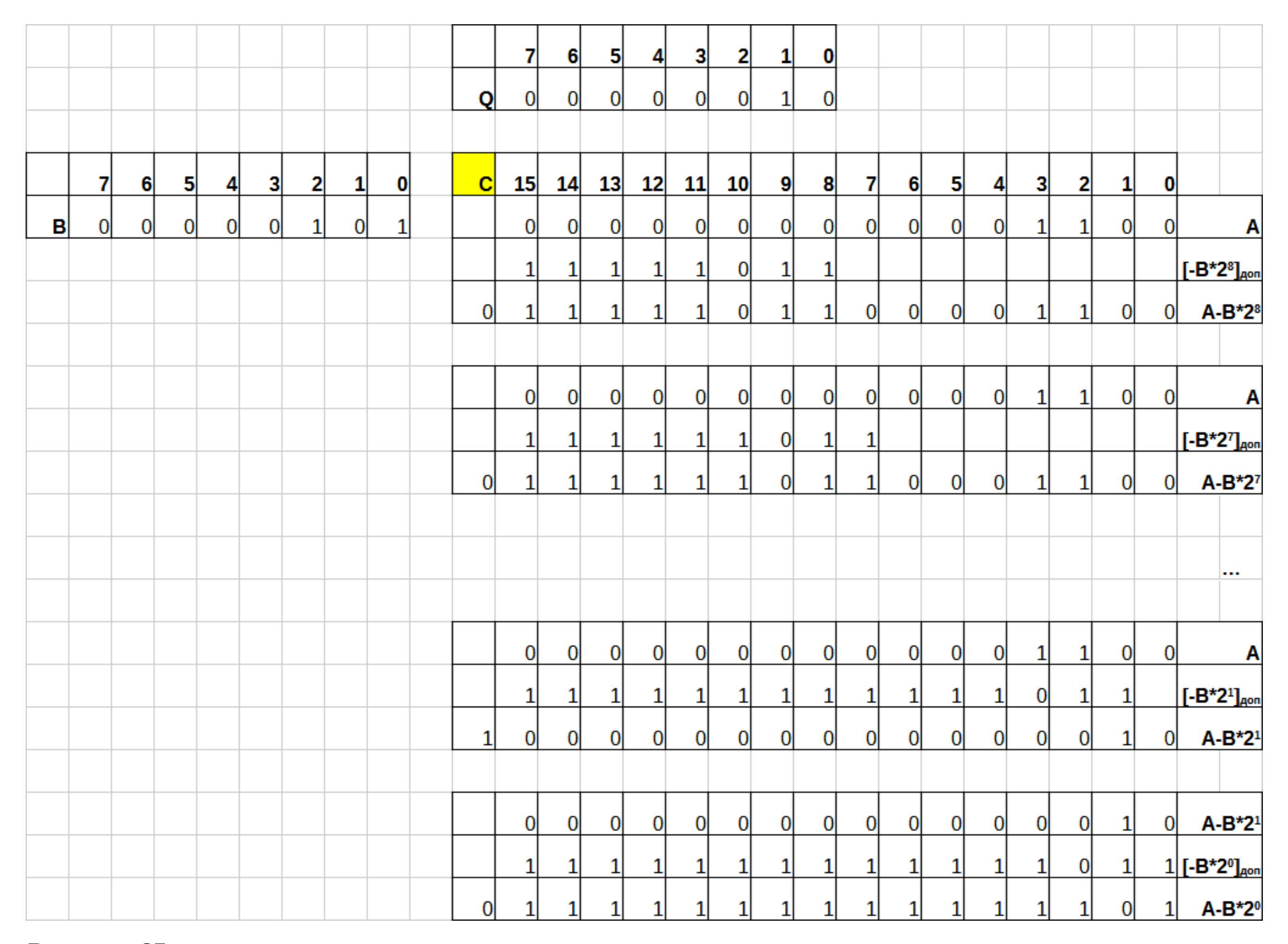
	7	6	5	4	3	2	1	0
A	0	0	0	0	1	1	0	0
B	0	0	0	0	0	1	0	1

Рисунок 64

Напомним проверку возможности деления:

$$\frac{A}{B}<2^8=256 \Rightarrow A<B*2^8 \Rightarrow A-B*2^8<0. \quad (16)$$

Далее выполним уже понятное нам целочисленное деление по базовой схеме.



	7	6	5	4	3	2	1	0
Q	0	0	0	0	0	0	1	0

	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	1

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	A
	1	1	1	1	1	0	1	1									$[-B*2^8]_{доп}$
0	1	1	1	1	1	0	1	1	0	0	0	0	1	1	0	0	$A-B*2^8$
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	A
	1	1	1	1	1	1	0	1	1								$[-B*2^7]_{доп}$
0	1	1	1	1	1	1	0	1	1	0	0	0	1	1	0	0	$A-B*2^7$
																	…
	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	A
	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1		$[-B*2^1]_{доп}$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	$A-B*2^1$
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	$A-B*2^1$
	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	$[-B*2^0]_{доп}$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	$A-B*2^0$

Рисунок 65

Мы получили неполное частное $Q=2$ и остаток $R=2$, действительно: $B*Q+R=5*2+2=12$. Пока что ничего нового — целочисленное деление с остатком.

Мы выяснили, что делитель $B=5$ укладывается целиком в делимом $A=12$ два раза. В остатке при этом остается 2, а поскольку $2<5$, то целое число раз делитель в остатке уже не укладывается, но он укладывается дробное число раз. То есть, мы можем выяснить, какая часть делителя укладывается в остатке: мы знаем, что он не укладывается один раз, но может, к примеру, он укладывается 0.9 или 0.5 раз?

Расширим сетку частного на один байт вправо дробными двоичными разрядами и вспомним, что максимальное значение дробной части, представимое в пределах байта, равно $2^{-8}*255=0.99609375$.

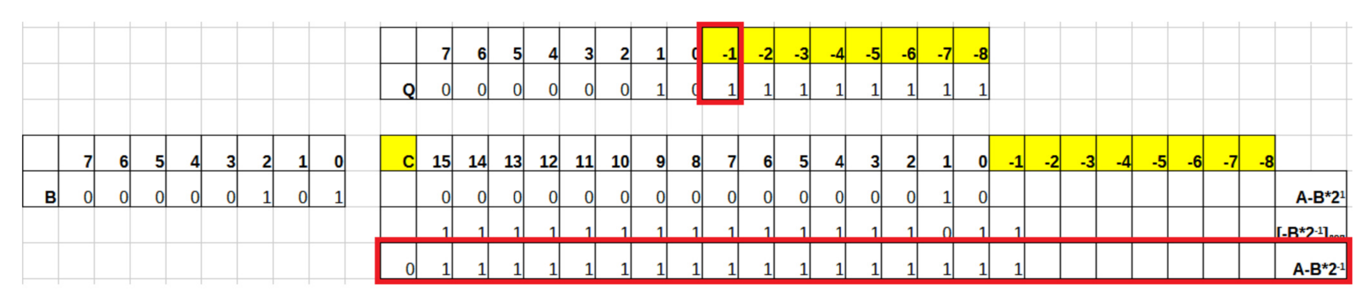
Далее, если в действительности делитель укладывается в остатке $R=2$, к примеру, $0.(9)$ раз, то значит он укладывается и 0.99609375. Поэтому, как и при целочисленном делении, мы допустим для начала, что делитель укладывается максимальное количество раз (мы уже знаем, что он не укладывается 1 раз, тогда предельный случай — это бесконечная десятичная дробь вида 0.99…,

но поскольку двоичная сетка ограничена, то мы вынуждены взять только максимальное представимое дробное значение).

Снова, если делитель укладывается $0.(9)$, то он укладывается и $0.99609375=0.5+0.49609375$ раз. То есть, делитель укладывается в том чисел и 0.5 раз (двоичный разряд с весом 2^{-1}).

Таким образом, как и в случае целочисленного деления, наша начальная гипотеза в том, что делитель укладывается максимальное дробное количество раз в полученном после целочисленного деления остатке. Поэтому последовательно умножаем делитель на веса дробных разрядов и вычитаем из остатка. Если наша гипотеза верна, то ни на одном шаге мы не получим отрицательный остаток.

Но поскольку мы будем умножать делитель на дробные веса, кроме сетки частного, нам также потребуется расширить вправо на 8 разрядов сетку делимого и делителя.



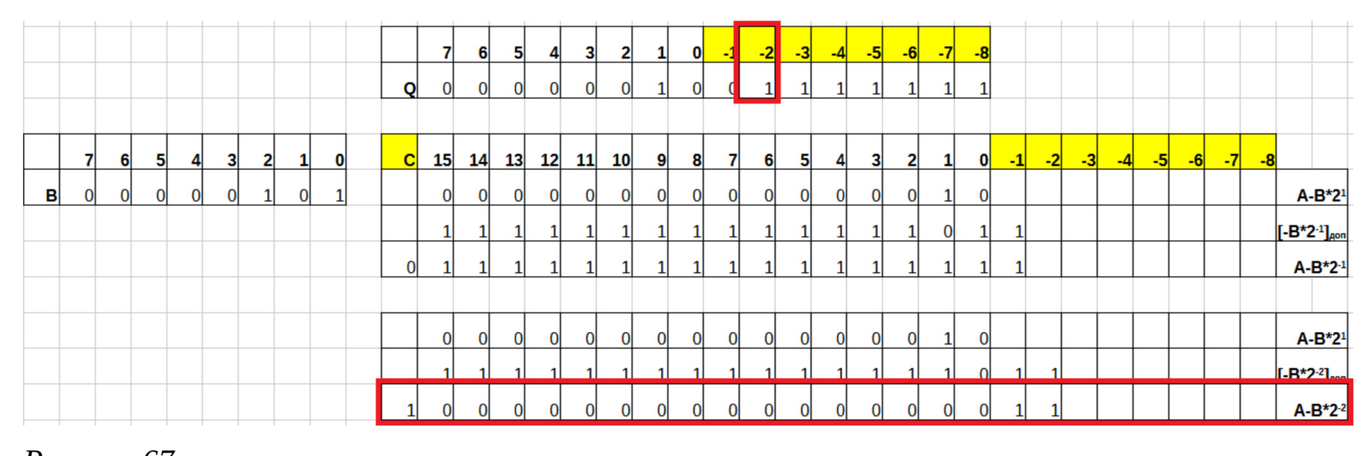
	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	1

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
Q	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0									$A-B*2^{1}$
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1								$[-B*2^{-1}]_{доп}$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1								$A-B*2^{-1}$

Рисунок 66

Мы умножили делитель на 0.5 и выяснили, что он не укладывается столько раз в остатке, тогда наша следующая гипотеза в том, что он укладывается $0.99609375-0.5=0.49609375$ раз (в действительности, в предельном случае, он может укладываться 0.4(9), но, опять же, т. к. сетка ограничена, то и мы ограничиваемся только максимальным представимым значением). А поскольку $0.49609375=0.25+0.24609375$, то проверяем следующий дробный разряд гипотетического частного с весом $2^{-2}=0.25$.



	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	1

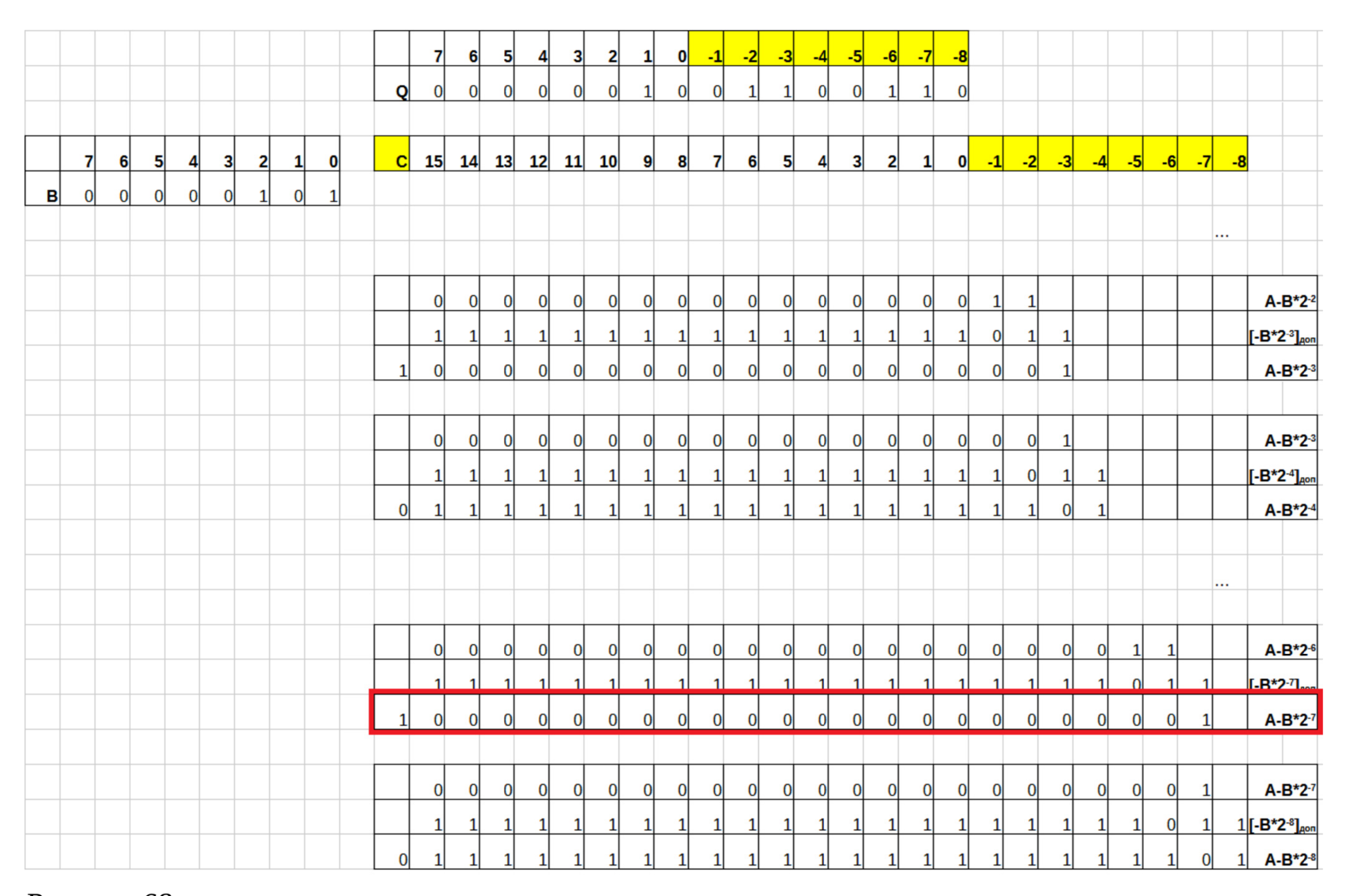
	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
Q	0	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0									$A-B*2^{1}$
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1								$[-B*2^{-1}]_{доп}$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1								$A-B*2^{-1}$
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0									$A-B*2^{1}$
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1							$[-B*2^{-2}]_{доп}$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1							$A-B*2^{-2}$

Рисунок 67

Мы выяснили, что делитель укладывается 0.25 раз. Еще 0.25 раз он уже не укладывается т. к. тогда он укладываелся бы 0.5 раз, поэтому следующая гипотеза после очередного ненулевого остатка в том, что он укладывается меньше 0.25, т.е. $(2^6-1)*2^{-8}=0.24609375=0.125+0.12109375=2^{-3}+0.12109375$.

Пропустим оставшиеся шаги и перейдем сразу к финальному результату.



	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
Q	0	0	0	0	0	0	1	0	0	1	1	0	0	1	1	0

	7	6	5	4	3	2	1	0
B	0	0	0	0	0	1	0	1

C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	
																							...		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1							$A-B*2^{-2}$
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1						$[-B*2^{-3}]_{доп}$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1						$A-B*2^{-3}$
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1						$A-B*2^{-3}$
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1					$[-B*2^{-4}]_{доп}$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1					$A-B*2^{-4}$
																							...		
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1			$A-B*2^{-6}$
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1		$[-B*2^{-7}]_{доп}$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		$A-B*2^{-7}$
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		$A-B*2^{-7}$
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	$[-B*2^{-8}]_{доп}$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	$A-B*2^{-8}$

Рисунок 68

Мы определили все дробные цифры частного в пределах байта. При этом остаток снова оказался ненулевым и равным остатку после целочисленного деления домноженному на 2^{-8}: $2*2^{-8}=2^{-7}$.

Теперь, если посмотрим на частное, то заметим, что дробь периодическая и если мы расширим сетку вправо еще на 8 дробных разрядов и продолжим вычисление цифр частного, то получим те же разряды частного (тот же паттерн) только с меньшими весами и в конце снова получим ненулевой остаток равный уже $2*2^{-8}*2^{-8}=2^{-15}$.

Таким образом, мы получили частное с точностью до 8 дробных двоичных разрядов, равное 2.3984375.

Если же разделить 12 на 5 в десятичной системе, то мы получим конечную десятичную дробь 2.4.

Рассмотрим число 2.5 в разложении в сумму $2+0.5$. Двойку мы сразу можем представить в двоичной системе счисления как $1*2^1=2$ (как и любое другое целое число). А теперь разложим $0.5=\frac{5}{10}=\frac{5}{2*5}=\frac{1}{2}$. То есть, 0.5 мы также можем представить в двоичной системе, достаточно единицу сдвинуть в разрядной сетке вправо на один разряд, чтобы получить $1*2^{-1}=\frac{1}{2}=0.5$. То есть, $2.5=1*2^1+0*2^0+1*2^{-1}$. Это означает, что число 2.5 разложимо в сумму степеней двойки.

А теперь рассмотрим 2.4. С двойкой все по-прежнему, она представима в двоичной системе. А 0.4 представим как $\frac{4}{10}=\frac{2*2}{2*5}=2/5$. То есть, дробная часть уже не является степенью двойки (ни положительной, ни отрицательной), следовательно, $\frac{2}{5}$ мы принципиально не можем представить в двоичном виде.

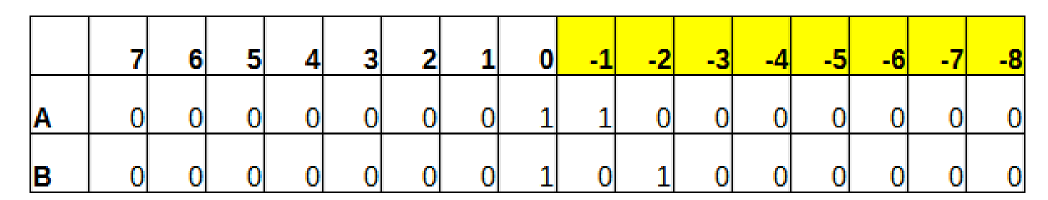
Двойка в дроби $\frac{2}{5}$ и есть остаток от целочисленного деления $12/5$. А когда мы расширяем сетку частного вправо до дробных разрядов и продолжаем деление, получая бесконечную периодическую дробную часть, мы получаем приближение к $\frac{2}{5}$. Понятно, что чем шире дробная часть частного, тем больше дробных разрядов частного мы вычислим и тем ближе станет наше частное к истинному частному 2.4.

NOTE: Мы выяснили, что число 2.4 не представимо точно в двоичном виде, то есть, истинное частное не разложимо в сумму степеней двойки. Это значит, что выполняя деление в бесконечной разрядной сетке мы никогда не получим нулевой остаток (иначе частное будет конечной двоичной дробью). Далее, делитель, домножаемый на веса цифр частного, всегда имеет фиксированную длину, то есть, все его двоичные ненулевые разряды сохраняются, только происходит постепенный сдвиг вправо. При этом, при вычитании делителя из текущего остатка (каким бы он ни был) в случаях, когда следующий остаток положительный (случай с отрицательным остатком мы всегда пропускаем и возвращаемся к последнему положительному) новый остаток по длине также оказывается ограничен длиной делителя, то есть, количество ненулевых двоичных разрядов очередного остатка оказывается не больше количества ненулевых двоичных разрядов делителя. Рассуждения здесь практически те же, что и при доказательстве безопасности сдвига влево, которое мы рассматривали в разделе, посвященном схеме деления без восстановления с неподвижным делителем: допустим, что старшие биты остатка и делителя совмещены, делитель при этом меньше остатка, тогда новый остаток окажется «в ненулевых» битах короче предыдущего. Если же делитель будет больше остатка, то мы сдвинем делитель вправо на один разряд (домножим на вес следующего разряда частного) и получим ситуация, когда старший бит остатка «нависает» над делителем, а младший бит делителя при этом выступает за пределами младшего бита остатка. При этом, как мы уже выясняли в доказательстве схемы с неподвижным без восстановления, бит нового остатка в позиции старшего бита текущего остатка окажется нулевым, при этом бит нового остатка в позиции младшего бита делителя (который выступает) окажется ненулевым, что по итогу снова даст

длину нового остатка не превышающую длину делителя. А поскольку мы достоверно знаем, что нулевый остаток никогда не будет получен, рано или поздно мы переберем все сочетания остатка и делителя и возникнет ситуация, когда очередной остаток окажется повторением того, который уже был и с этого момента, если мы продолжим деление, цифры частного вновь начнут повторяться. То есть, раз мы получим тот же остаток, который мы уже получали, значит мы получим ту же разность, которую мы уже получали (делитель ведь не меняет своего паттерна), а значит мы получим ту же цифру частного (в зависимости от знака разности), которую мы уже получали, только с меньшим весом, то есть, дальше в сетке справо.

Деление дробных

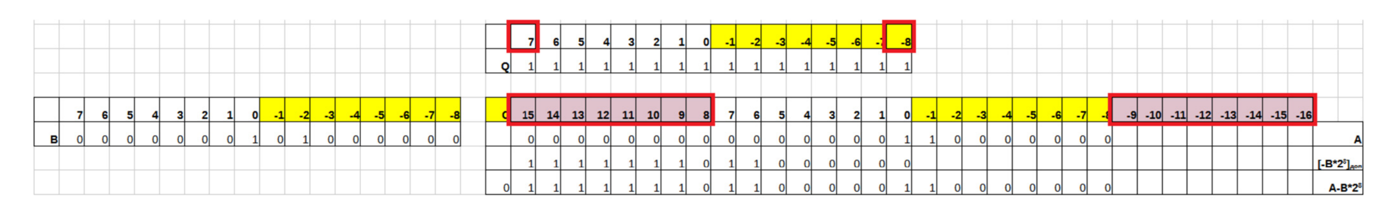
Возьмем два числа: $A=1.5$, $B=1.25$ и разделим их. Под частное отведем два байта: байт под целую и байт под дробную.



	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0

Рисунок 69

При этом, поскольку веса MSB и LSB частного равны 2^7 (плюс проверка возможности деления на первом шаге: умножение на 2^8) и 2^{-8} соответственно, нам необходимо расширить целую и дробную части сетки делимого на байт, т. е. суммарно — на два байта (выделено фиолетовым на рисунке).



	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
B	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
Q	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0									A
	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0																	$[-B*2^8]_{доп}$
0	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0									$A-B*2^8$

Рисунок 70

После деления непосредственно в дробных мы получаем неполное частное 1.19921875 и остаток 0.0009765625 (1.25∗1.19921875+0.0009765625=1.5. На рисунке, для экономии места, опущено большинство случаев отрицательного остатка.

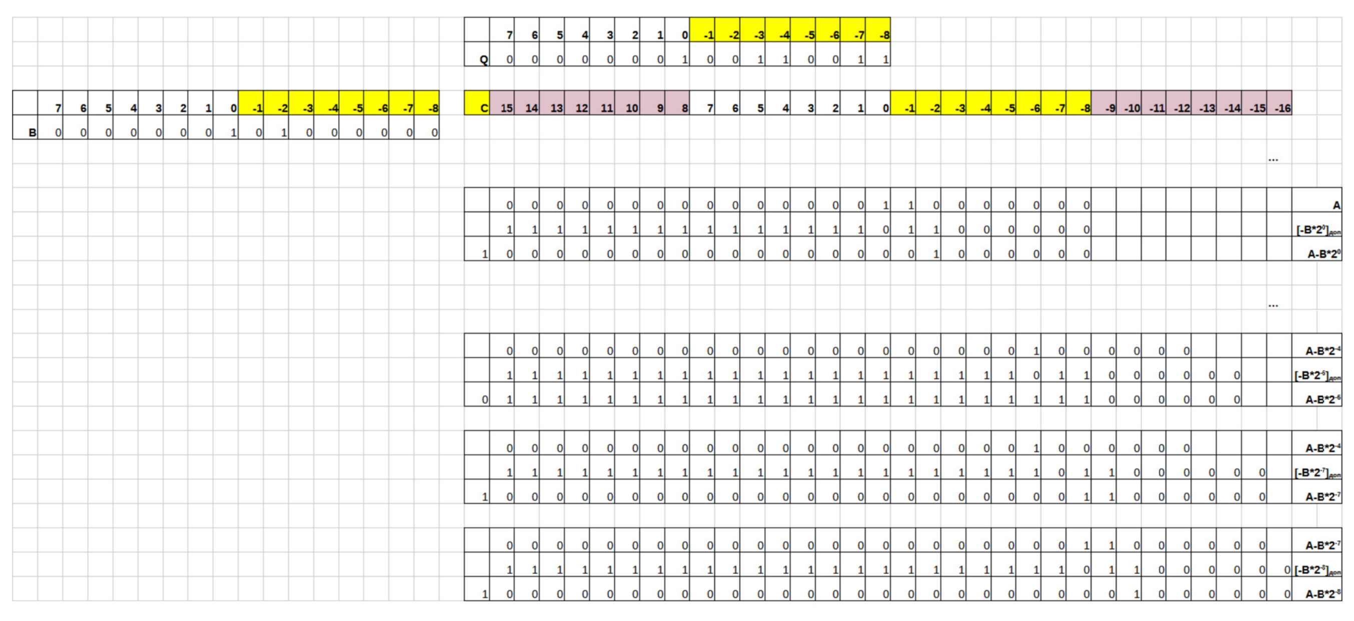


Рисунок 71

Истинное же частное в десятичной системе равно 1.2. Но как мы уже выше разбирали: $0.2=\frac{2}{10}=\frac{1}{5}$, т. е. дробная часть непредставима точно в двоичном виде и результатом деления будет бесконечная периодическая двоичная дробь.

Истинное частное не представимо точно, а значит мы никогда не получим нулевой остаток. При этом, поскольку длина делителя фиксирована, то длина остатка (в смысле количество ненулевых единичных разрядов) также не превысит длины делителя, а это значит, что значения получаемых остатков неизбежно будут повторяться (имеются ввиду паттерны единиц и нулей, а не абсолютные значения остатков, абсолютные значения, конечно, будут разными, т. к. будут иметь все меньшие и меньшие значения, располагаясь справа в разрядной сетке), а значит будут повторяться и цифры частного. Это мы и получаем в данном примере.

Переход к целым, случай 1

Возьмем те же числа: $A=1.5$, $B=1.25$. Но предварительно отмасштабируем их, перейдя от дробных к целым, избавившись таким образом от дробной части: $A=1.5*2^8=384$, $B=1.25*2^8=320$. Понятно, что масштабирование не изменит истинного частного: $\frac{A*2^8}{B*2^8}=\frac{A}{B}=1.2$. Частное будем вычислять в той же самой сетке размером в два байта, где младший байт отведен под дробную часть (легко проверить, что несмотря на масштабирование операндов, деление с заданной сеткой частного возможно).

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
A'	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
B'	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0

Рисунок 72

Далее, поскольку MSB частного имеет вес 2^7 и при этом мы выполняем проверку возможности деления, умножая делитель на 2^8 на первом шаге, а отмасштабированный делитель B по-прежнему занимает два байта, то сетку отмасштабированного делимого также, как и в случае с делением дробных, необходимо расширить влево на один байт. Ну а поскольку LSB частного имеет вес 2^{-8}, то нам также требуется расширение сетки влево еще на один байт. Таким образом, как и при делении непосредственно в дробных, мы расширяем сетку делимого влево и вправо по байту.

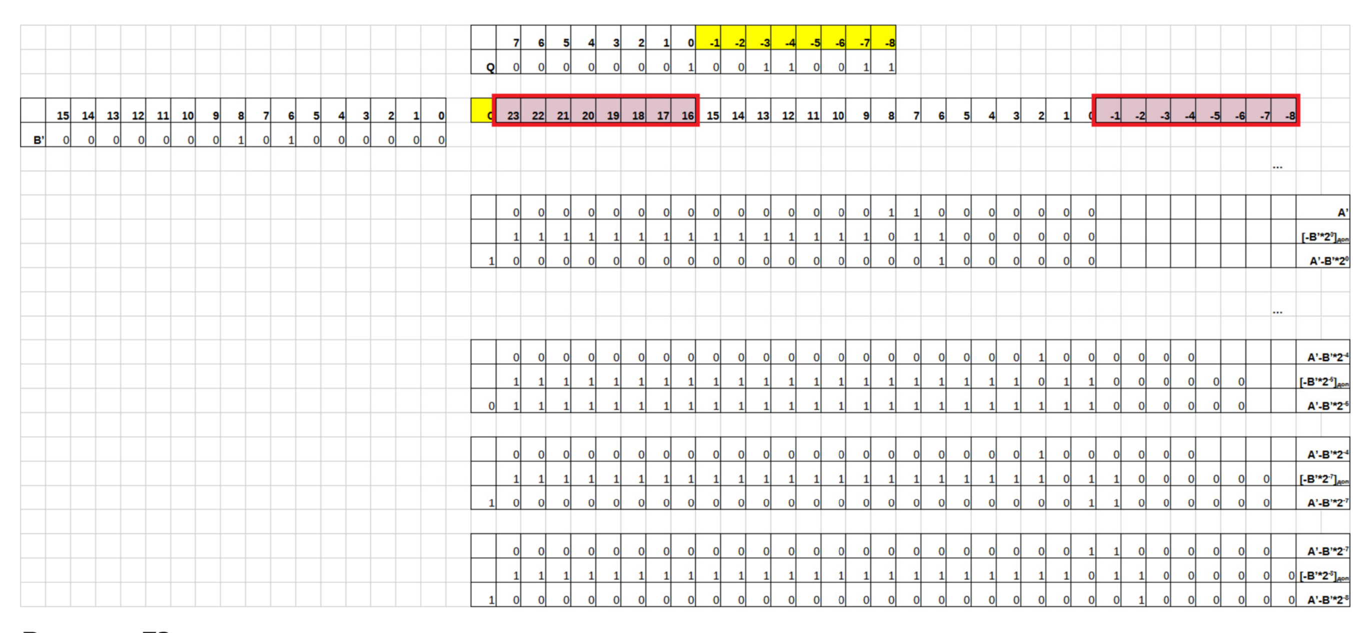


Рисунок 73

Как видим, технически, масштабирование операндов никак не изменило схему деления. Изменились только абсолютные значения операндов и, соответственно, остатков.

Когда мы делили дробные операнды, мы вычисляли на каждом шаге новый остаток: $A-B*Q[i]$, где $Q[i]$ — вес очередной цифры частного. Сейчас же мы отмасштабировали исходные операнды: $A*k$ и $B*k$, где $k=2^8$ — масштабный коэффициент. Поэтому сейчас мы вычисляем на каждом шаге остаток следующим образом: $A*k-B*k*Q[i]=k*(A-B*Q[i])$. То есть, на каждом шаге мы получаем тот же самый, что и при делении дробных, остаток, только отмасштабированный на k. Именно поэтому схема деления оказывается полностью эквивалентной делению непосредственно в дробных, т. к. цифра частного определяется не абсолютной величиной нового остатка, а его знаком, а знак не меняется, поскольку не меняется исходное отношение между отмасштабированными операндами.

Переход к целым, случай 2

Повторим переход к целым из предыдущего раздела, но число A домножим еще раз на 2^8: $A=1.5*2^8*2^8=1.5*2^{16}=98304$, $B=1.25*2^8=320$.

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
A	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
A'	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B'	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0

Рисунок 74

Тогда истинное частное будет равно $\frac{A*2^{16}}{B*2^8}=\frac{1.5*2^8*2^8}{1.25*2^8}=\frac{1.5*2^8}{1.25}=\frac{1.5}{1.25}*2^8=\frac{A}{B}*2^8$, т. е. будет отмасштабировано коэффициентом 2^8. А поскольку деление в данном случае целочисленное, то результатом будет неполное частное $Q=307$ и остаток $R=64$. То есть, результатом будет приближенное частное (с точностью до 8 дробных разрядов), отмасштабированное коэффициентом 2^8 и будет целым числом, а не дробным, поэтому переж вычислением двойную сетку частного также сдвинем вправо, избавившись от дробной части.

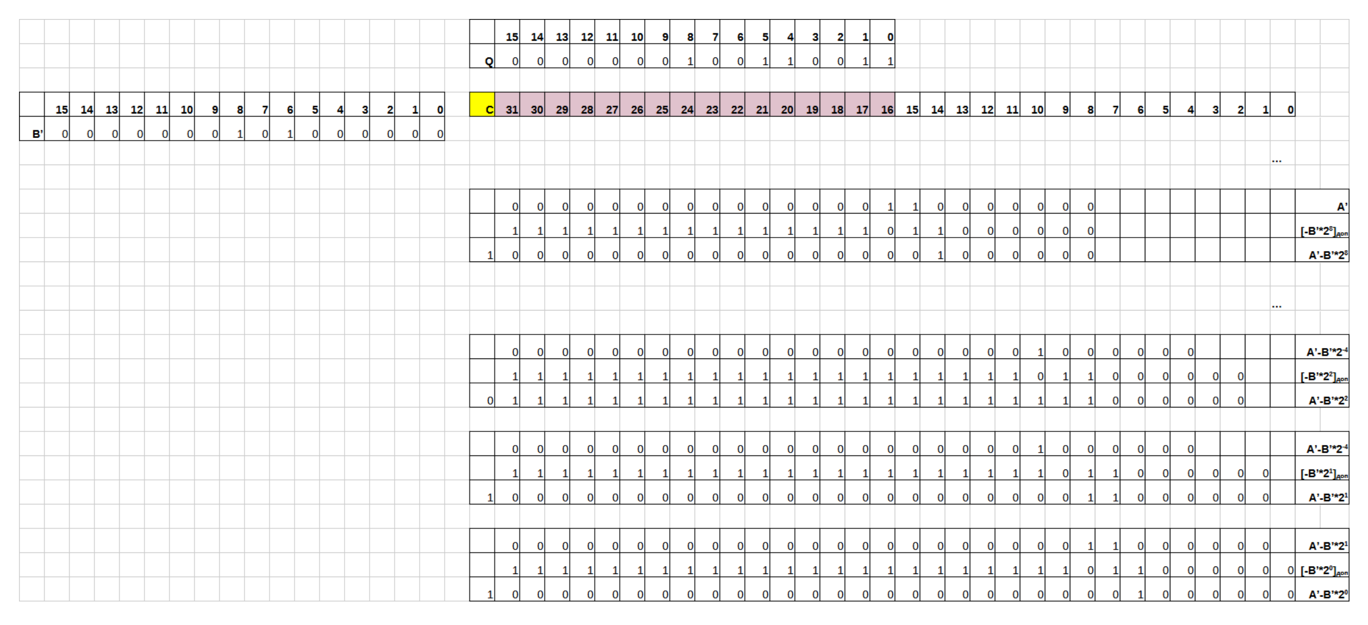


Рисунок 75

Здесь мы перешли полностью к целочисленной схеме деления и убедились, что технически схема снова не изменилась, поменялись только весовые коэффициенты разрядов и для возврата к истинному результату достаточно только мысленно скорректировать сетку частного, сдвинув ее влево на 8 разрядов.

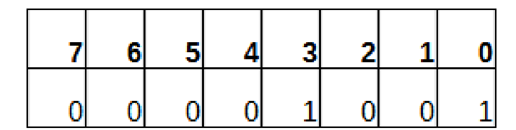
Таким образом, при дробном делении мы всегда можем перейти к полностью целочисленной и эквивалентной (в смысле выполняемых действий над битами, безотносительно их весов) схеме. А это значит, что мы можем применить уже обоснованную нами схему с неподвижным делителем без восстановления остатка. Более того, остается справедливым и обоснование достаточности расширения сетки влево на один разряд для безопасного сдвига остатка влево (как доп. кода так и положительного значения) в случае схемы с неподвижным делителем. Но понятно, что переход к целым не обязателен, мы можем просто повторить те же самые рассуждения, оставаясь в дробных числах, разница будет лишь в абсолютных значениях весовых коэффициентов разрядов, отношения же между весами разрядов не изменятся.

Программная эмуляция плавающей точки

Представление чисел в формате IEEE 754

TODO: Рассказать о спец. значениях и о денормализованных числах.

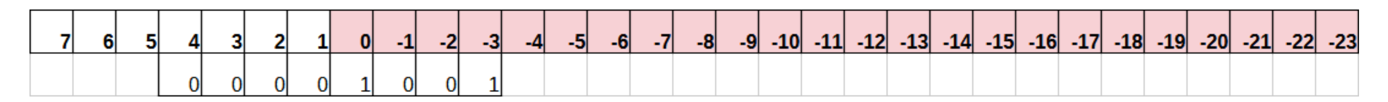
Возьмем число $A=9$ и запишем его в двоичном виде в пределах байта.



7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1

Рисунок 76

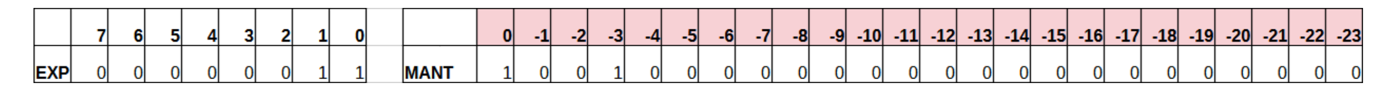
Затем расширим разрядную сетку вправо на 23 дробных разряда и будем делить число A на 2, т. е. сдвигать вправо, до тех пор, пока его значение не станет меньше 2.



7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
			0	0	0	0	1	0	0	1																				

Рисунок 77

Всего потребовалось 3 сдвига вправо. Сохраним количество сдвигов в счетчик EXP размером в байт, запоминая, таким образом, на какую степень двойки мы поделили исходное число, для того, чтобы знать, на какую степень двойки нужно умножить результат, чтобы вернуться к исходному значению. То есть, EXP — это двоичая экспонента.



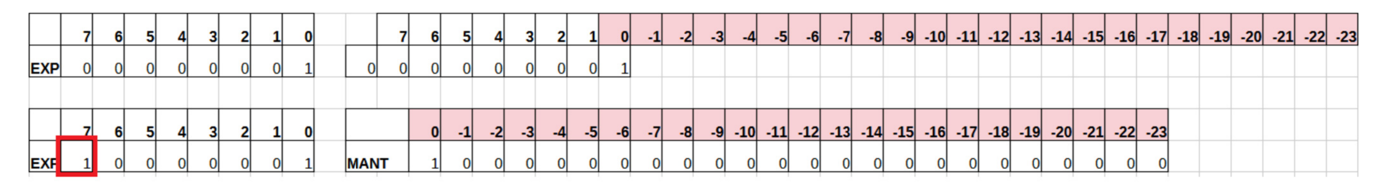
	7	6	5	4	3	2	1	0
EXP	0	0	0	0	0	0	1	1

	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
MANT	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 78

У самого же результата отбросим нулевые целочисленные разряды. По сути, мы нормализовали число A двоично, уместив его в отрезке $[2,1]$. При этом мы сохранили истинный порядок исходного числа равный 2^3. То есть, истинное значение A теперь выражается через произведение нормализованной мантиссы и двоичной экспоненты: $1.125*2^3$.

Теперь возьмем число $B=0.5$ и тоже нормализуем его, сформировав двоичную экспоненту.



	7	6	5	4	3	2	1	0
EXP	0	0	0	0	0	0	0	1

	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
0	0	0	0	0	0	0	0	1																							

	7	6	5	4	3	2	1	0
EXP	1	0	0	0	0	0	0	1

	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
MANT	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

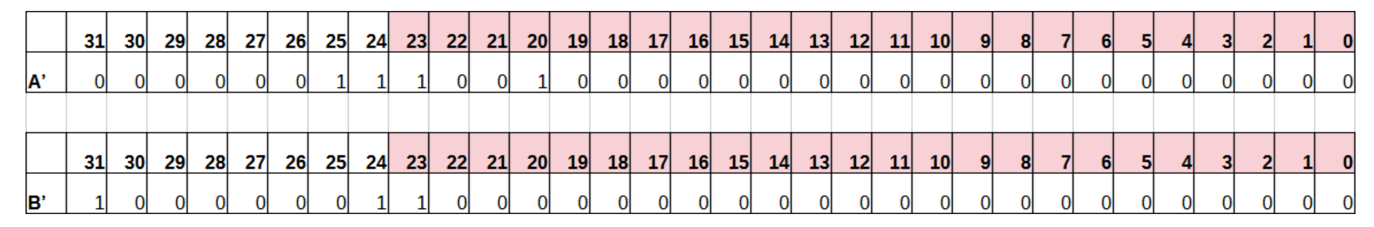
Рисунок 79

На этот раз для нормализации потребовался сдвиг влево на один разряд, т. е. умножение на 2, следовательно, экспонента равна единице, но с отрицательным знаком, т. к. для возврата к истинному значению необходимо мантиссу разделить на 2: $B=1*2^{-1}$. Отрицательную экспоненту пока запишем в прямом коде (sign-magnitude).

Экспонента в коде со смещением.

TODO: Показать еще одно преимущество кода со смещением, которое заключается в том, что при переполнении мантиссы, происходит автоматический инкремент экспоненты (а в граничном случае — автоматическое формирование inf).

Теперь возьмем числа, полученные в предыдущем разделе, склеим экспоненты и мантиссы вместе и будем рассматривать новые числа A' и B' как целые:



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A’	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B’	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 80

Конечно, A и B могут быть как положительными так и отрицательными и нам следовало бы добавить еще один разряд слева для знака (чтобы представлять отрицательные значения чисел A и B, например, в прямом коде). Но для наших целей этого не нужно, мы будем рассматривать только модули.

Поставим теперь общую задачу сравнения этих чисел между собой. Заметим следующее:

- Если знаки чисел разные, то больше будет то число, знак которого положительный.
- Если же знаки одинаковые и отрицательные, то больше будет то, модуль которого меньше.
- Если же знаки одинаковые и положительные, то больше будет то, модуль которого больше.

В первом случае сравнение сводится только к проверки знаков чисел. В последних двух случаях сравниваются модули.

Это была общая задача сравнения чисел, с учетом знака. Мы же будем сравнивать между собой только модули чисел. Нам уже очевидно, что число $A=9$ по модулю больше числа $B=0.5$.

Вспомним, что экспонента числа B отрицательная и рассмотрим сразу оба варианта: когда отрицательная экспонента представлена в прямом коде и в доп. коде.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A’	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B’	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B’’	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 81

Видим, что в этом случае сравнивать модули чисел между собой, оперируя целочисленной интерпретацией склейки экспоненты и мантиссы, невозможно, т. к. целочисленная интерпретация числа B оказывается больше A как в прямом так и в доп. коде.

Вспомним, что в прямом коде в пределах байта мы можем представить значения экспоненты в отрезке $[-127,127]$. В доп. коде — в отрезке $[-128,127]$.

Мы же остановимся на представлении значений экспоненты только в отрезке $[-127,128]$, но каждый раз, когда захотим сохранить экспоненту, будем прибавлять смещение 127. Таким образом, хранимые значения экспоненты будут положительными значениями в отрезке $[0,255]$. При этом, отношение экспонент сохранится: например, если $\mathrm{EXP}_1=-1<\mathrm{EXP}_2=3$, то $\mathrm{EXP}_1+127=126<\mathrm{EXP}_2+127=130$.

TODO: Почему выбран именно этот диапазон: [-127,128]? Сослаться на Кахана?

TODO: Отметить, что -127 и 128 зарезервированы под inf/nan и денормализованные числа.

Теперь вернемся к нашему примеру, но представим экспоненты уже в коде со смещением 127.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A’	1	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B’	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 82

Желтым выделена зона экспоненты. Видно, что теперь мы можем сравнивать модули чисел между собой, сравнивая склейки экспонент в коде со смещением и мантисс. Если порядки равны, то больше будет то число, мантисса которого больше, если же, например, порядок A больше порядка B, но при этом мантисса A меньше мантиссы B, то сравнение склеек A' и B' между собой даст корректный результат и целочисленная интерпретация склейки экспоненты и мантиссы A' окажется больше B'. Главное — выполняя в дальнейшем арифметические действия с экспонентой помнить, что мы работаем не с истинным значением, а с EXP+127, где EXP — истинное значение экспоненты.

Диапазон представимых значений

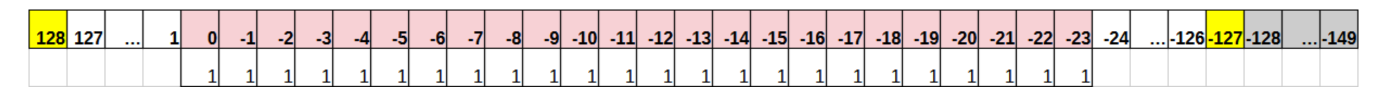
Мы условились, что в пределах байта будем представлять экспоненты из отрезка $[-127{,}128]$. Посмотрим, какие значения мы можем представлять с таким ограничением на экспоненту и с ограничением на мантиссу в 24 разряда.

Здесь мы должны сразу сделать оговорку: в стандарте IEEE 754 значения экспонент -127 и 128 (0 и 255 в коде со смещением) не используются для представления чисел, они зарезервированы под т. н. спец. значения, о которых мы поговорим позже. То есть, когда экспонента равна -127 или 128, стандартная числовая интерпретация $MANT*2^{\mathrm{EXP}}$ не применяется, а применяется другая.

Возьмем за основу максимальное нормализованное число меньше двух: $(2^{24}-1)*2^{-23}$. Экспонента его равна нулю (или 127 в коде со смещением). Теперь выясним наибольшее и наименьшее значения, которые мы можем представить, учитывая ограничения на экспоненту и на мантиссу. Максимальное значение будет равно $(2^{24}-1)*2^{-23}*2^{127}$ а минимальное — $(2^{24}-1)*2^{-23}*2^{-126}$.

Посмотрим на эти числа в пределах неограниченной разрядной сетки, т. е. с «раскрытыми» экспонентами.

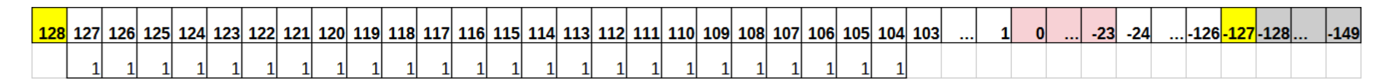
Это нормализованное число $(2^{24}-1)*2^{-23}$:



128	127	…	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	…	-126	-127	-128	…	-149
				1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1							

Рисунок 83

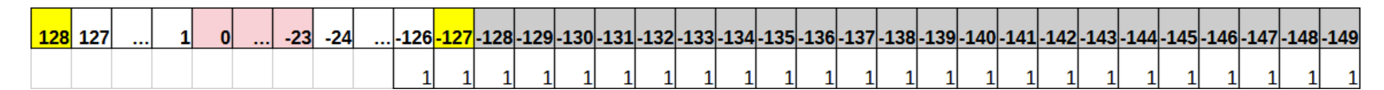
Это максимальное значение, представимое в одинарном float (помним, что значения экспонент 128 и -127 зарезервированы под спец. значения):



128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113	112	111	110	109	108	107	106	105	104	103	…	1	0	…	-23	-24	…	-126	-127	-128	…	-149
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1													

Рисунок 84

А это минимальное значение:



128	127	…	1	0	…	-23	-24	…	-126	-127	-128	-129	-130	-131	-132	-133	-134	-135	-136	-137	-138	-139	-140	-141	-142	-143	-144	-145	-146	-147	-148	-149
									1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Рисунок 85

На самом деле IEEE 754 даёт возможность представлять значения меньше указанного нами наименьшего: для это вводится понятие денормализованного числа. Особенность в том, что мы сохраняем мантиссу в денормализованном виде (то есть, меньше единицы) и устанавливаем в поле экспоненты значение -127 ($-127+127=0$ в коде со смещением) как признак того, что мантиссу нужно интерпретировать как число меньше единицы. Но в калькуляторе мы денормализованные числа не реализуем, поэтому поговорим об этом вкратце потом, отдельным разделом.

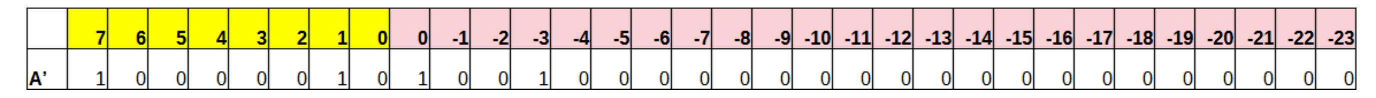
TODO: Сделать пошаговый инкремент, чтобы показать, что с ростом экспоненты растёт разреженность, т. е. величина шага.

Упаковка

Итак, числа в формате плавающей точки (одинарной точности) представлены двумя частями: нормализованная мантисса длиной в 24 разряда и экспонента в коде со смещением длиной в байт (т. е. всего $24+8=32$ байта).

Но стандарт IEEE 754 определяет т. н. упакованный формат представления, в котором целочисленная единица мантиссы не хранится, но подразумевается (за исключением случая, когда число является денормализованным).

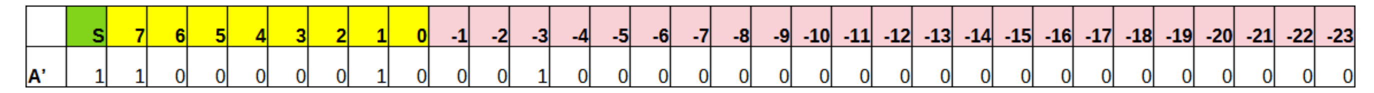
Возьмем число $A = -9$, модуль которого уже представлен экспонентой в коде со смещением и нормализованной мантиссой: $MANT = 1.125$, $EXP = 3 + 127 = 130$.



	7	6	5	4	3	2	1	0	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
A'	1	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 86

Теперь сдвинем экспоненту вправо на один разряд так, чтобы младший бит экспоненты перезаписал старший бит мантиссы, который представляет целочисленную единицу.

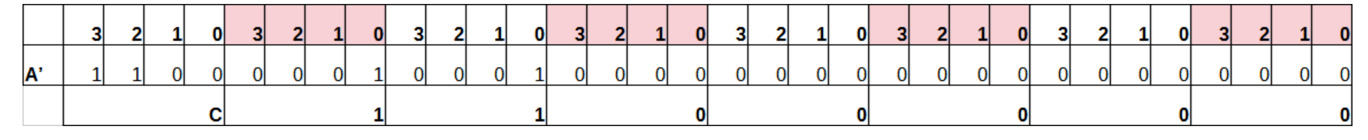


	S	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
A'	1	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 87

Длина разрядной сетки осталась прежней — 32 разряда, но мантисса теперь занимает только 23 бита, при этом самый старший бит освободился (выделен зеленым) и мы можем записать в него знак числа A в прямом коде.

Разобьем полученную цепочку на байты и переведем каждый байт в hex, получив таким образом шестнадцатеричное представление числа $A = -9$ в формате float одинарной точности:



	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
A'	1	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
				C				1				1				0				0				0				0				0

Рисунок 88

Для большей убедительности можем свериться с сишным кодом:

```
#include <iomanip>
#include <iostream>
#include <cstdint>

void print_float_as_hex(float* a) {
    uint32_t int_rep = *(uint32_t*)a;
    std::cout << std::hex << std::setfill('0') << std::setw(8) << int_rep << std::endl;
}

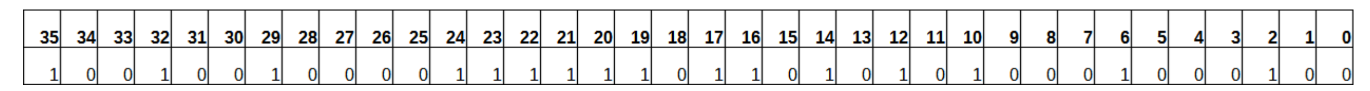
int main() {
    float a = -9.0f;
    print_float_as_hex(&a);

    return 0;
}
```

Округление

В меньшую сторону

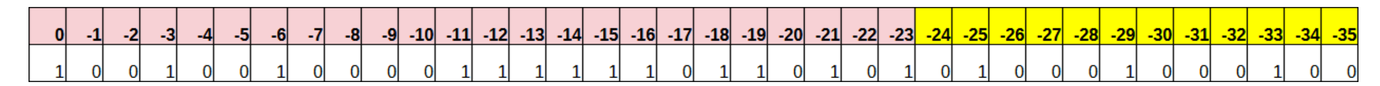
Возьмем целое число $A=39224824900$ и представим его в формате плавающей точки одинарной точности.



35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	0

Рисунок 89

Для нормализации потребуется 35 сдвигов вправо, т. е. двоичная экспонента $EXP=35$ или в коде со смещением: $EXP'=35+127=162$.



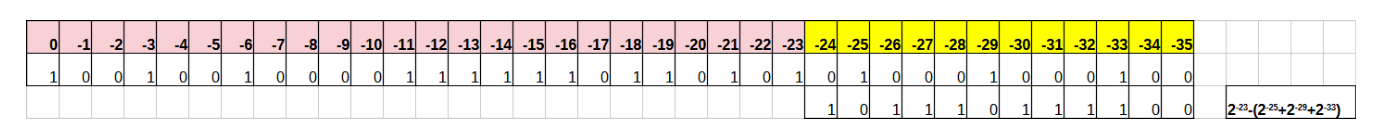
0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27	-28	-29	-30	-31	-32	-33	-34	-35
1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	0

Рисунок 90

Нормализованная истинная мантисса равна 1.141592653584666550159454345703125. При этом мы видим, что она не вмещается в 24 разряда и имеет хвост справа, равный $0.31781382858753204345703125*10^{-7}$, т. е. возникает задача округления мантиссы перед упаковкой.

Если мы просто отбросим хвост, то получим значение на числовой оси слева от истинного, равное 1.14159262180328369140625 и модуль ошибки $|ERR|=0.31781382858753204345703125*10^{-7}$, которая меньше $2^{-24}=\frac{2^{-23}}{2}$, т. е. меньше половины веса последнего разряда мантиссы длиной в 24 бита (с учетом экспоненты, эта величина известна как ULP/2: $\frac{(2^{-23}*2^{EXP=35})}{2}$).

Если же мы прибавим $2^{-23}-(2^{-25}+2^{-29}+2^{-33})$ (то есть, значение, которое даст зануление «хвоста» и бит переноса в область представимой в 24 разрядах мантиссы) то получим значение уже справа от истинного, равное 1.1415927410125732421875 и модуль ошибки $|ERR|=0.87427906692028045654296875*10^{-7}$, которая (после домножения на экспоненту) будет уже больше половины ULP.



0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27	-28	-29	-30	-31	-32	-33	-34	-35		
1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0	1	0	0	0	1	0	0		
																								1	0	1	1	1	0	1	1	1	1	0	0		$2^{-23}-(2^{-25}+2^{-29}+2^{-33})$

Рисунок 91

Очевидно, что в нашем случае, для округления к ближайшему числу, которое представимо в пределах 24 разрядов, нужно отбросить хвост (по тому же принципу, по которому выполняется округление в десятичной системе).

Таким образом, при упаковке в одинарный float числа $A=39224824900$ мы получили число $A'=1.14159262180328369140625*2^{35}=39224823808$ и абсолютную ошибку равную $A'-A=-1092$.

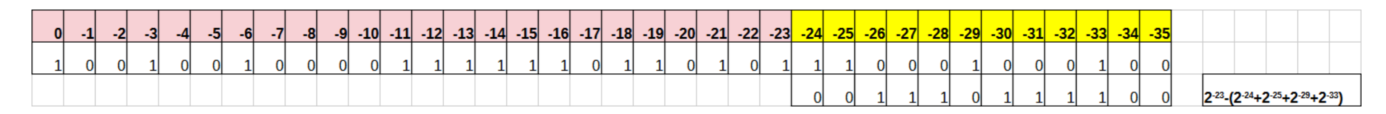
Для убедительности можно свериться с аппаратной реализаций на десктопе:

```
#include <iomanip>
#include <iostream>

int main() {
    float a = 39224824900.0f;
    std::cout << "fixed: " << std::fixed << std::setprecision(50) << a << "\n";
    return 0;
}
```

В большую сторону

Возьмем теперь число $A=39224826948$ и также упакуем в одинарный float.



0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27	-28	-29	-30	-31	-32	-33	-34	-35		
1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	0	1	1	1	0	0	0	1	0	0	0	1	0	0		
																								0	0	1	1	1	0	1	1	1	1	0	0		$2^{-23}-(2^{-24}+2^{-25}+2^{-29}+2^{-33})$

Рисунок 92

Видно, что если мы отбросим «хвост», то ошибка по модулю будет больше половины ULP, поэтому для округления к ближайшему числу мы прибавляем $2^{-23}-(2^{-24}+2^{-25}+2^{-29}+2^{-33})$, что дает ошибку меньше половины ULP.

Конкретное значение, которое нужно прибавить, чтобы занулить «хвост», в случае ситуации округления в большую сторону, необязательно каждый раз вычислять явно. Достаточно посмотреть на разряда с весом 2^{-24} и, если он ненулевой, отбросить «хвост» и прибавить 2^{-23} (кроме ситуации, когда «хвост» представлен только единицей в позиции -24, эту ситуацию мы рассмотрим далее отдельно).

Действительно, если обозначить нормализованную мантиссу как A, и отбрасываемый «хвост», который больше половины ULP, обозначить как $B=2^{-24}+b$, где b — это сумма весов ненулевых разрядов после разряда с весом 2^{-24}, то легко показать, что $A-B+2^{-23}=A+(2^{-23}-B)$.

Halfway

И, наконец, возьмем последнее число: $A = 39224825856$.

0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27	-28	-29	-30	-31	-32	-33	-34	-35		
1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0		
																								1	0	0	0	0	0	0	0	0	0	0	0		2^{-24}

Рисунок 93

Здесь мы получили симметричную ситуацию округления. То есть, независимо от того, отбросим мы «хвост», равный 2^{-24}, или прибавим $2^{-23} - 2^{-24} = 2^{-24}$, мы получим одинаковую по модулю ошибку равную половине ULP. Стандар IEEE 754 определяет для такой ситуации стратегию округления к ближайшему/к четному (to nearest/ties to even). Это означает, что если ULP содержит единицу, то мы округляемся в большую сторону, прибавляя 2^{-24} и получая четное значение. Если же ULP уже равен нулю, то округляемся в меньшую сторону, отбрасывая хвост также получая четное значение.

В нашей ситуации, для получения четного значения, требуется округление в большую сторону, что дает в итоге число $A' = 39224827904$.

NOTE: На уровне реализации достаточно прочитать ULP, отбросить «хвост» и прибавить прочитанный ULP.

Реализация умножения — FMUL

TODO: Сказать о компактной схеме с округлением частичных произведений.

Возьмем числа $A=253530089822500389933611273420 8$ и $B=100663296$. Оба числа точно представимы в одинарном float, т. е. не требуют округления.

Далее, для удобства будем рассматривать их двоичные экспоненциальные представления: $A=1.99999976158142089843 75*2^{100}$; $B=1.5*2^{26}$. Тогда истинное произведение этих чисел в экспоненциальной же форме будет равно:

$$1.9999997615814208984375*2^{100}*1.5*2^{26}=1.9999997615814208984375*1.5*2^{100+26}= \\ = 2.99999964237213134765625*2^{126}. \quad (17)$$

Начнем с вычисления экспоненты произведения. В коде со смещением $EXP(A)=100+127=227$, $EXP(B)=26+127=153$. Складываем экспоненты: $(100+127)+(26+127)=(126+127)+127=380$. Получили истинную экспоненту в коде со смещением плюс перебор равный 127. Поэтому вычитаем 127: $EXP(A*B)=126+127=253$.

Далее перемножим мантиссы по уже известной нам схеме с неподвижным множимым: с множимым и аккумулятором частичных произведений свяжем подвижную виртуальную сетку (но поскольку первое частичное произведение образовано разрядом множителя с весом 2^{-23}, а вес наименьшего значащего разряда множимого тоже равен 2^{-23}, то веса значащих разрядов виртуальной двойной сетки будут лежать в отрезке $[2^1=2, 2^{-23}*2^{-23}=2^{-46}]$), складывать будем только старшие части множимого и аккумулятора, на каждой итерации будем сдвигать аккумулятор вправо вместе с подвижной виртуальной сеткой.

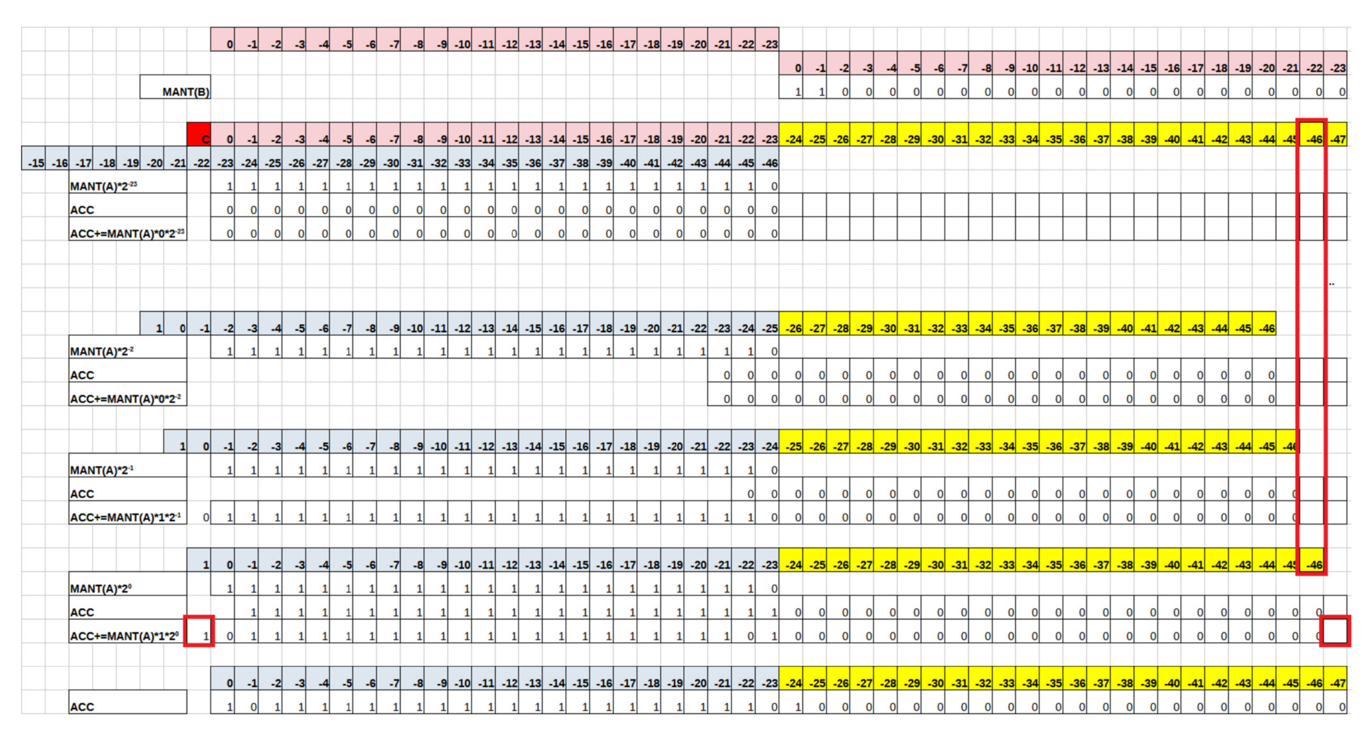


Рисунок 94

На последней итерации виртуальная разрядная сетка встает вровень с истинной сеткой, поэтому на последнем шаге всегда имеем истинное произведение.

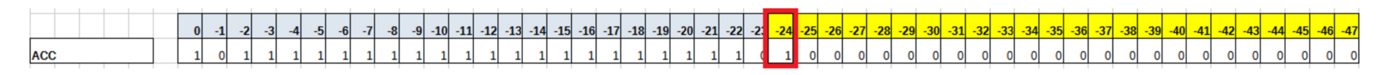
Но относительно истинной сетки в данном примере мы получили бит переноса, то есть, денормализацию произведения влево, поэтому требуется нормализация вправо. При этом на рисунке видно, что разряд истинного произведения с весом 2^{-47} всегда будет равен нулю, поэтому нормализация вправо всегда безопасна. Нормализуем и корректируем экспоненту, прибавляя единицу.

В итоге получаем нормализованную мантиссу истинного произведения в двойной разрядной сетке и скорректированную экспоненту в коде со смещением:

$$MANT(A*B)*2^{-1}=1+(2^{21}-1)*2^{-22}+2^{-24}=1.499999821186065673828125.$$
$$EXP(A*B)+1=126+127+1=253+1=254. \quad (18)$$

Действительно,
$1.49999982118606567382812 5*2^{(126+1=127)}=255211744767089442120025319652949229568$, что в точности равно истинному произведению:
$1.99999976158142089843 75*1.5*2^{100+26=126}=255211744767089442120025319652949229568$.

С экспонентой все хорошо: в коде со смещением она равна $127+127=254$ и представима в формате одинарной точности. А вот мантисса произведения не вмещается в 24 разряда, т. к. имеет ненулевой разряд с весом 2^{-24} за пределами сетки одинарной точности:



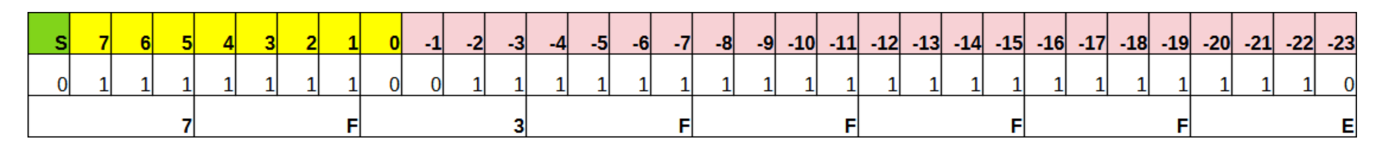
	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
ACC	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

	-24	-25	-26	-27	-28	-29	-30	-31	-32	-33	-34	-35	-36	-37	-38	-39	-40	-41	-42	-43	-44	-45	-46	-47
ACC	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 95

Вспоминая раздел про округление мы видим, что перед нами ситуация симметричного округления (halfway). То есть, чтобы занулить «хвост» мы можем как отбросить 2^{-24} так и прибавить — ошибка по модулю будет одинаковой. Но ULP уже равен нулю, поэтому в соответствии с дефолтной схемой округления (к ближайшему/к четному), определенной стандартом, мы просто отбрасываем 2^{-24} и получаем четное значение округленной мантиссы.

В итоге, после округления мантиссы и представления экспоненты в коде со смещением, финальный результат произведения двух чисел $A=2535300898225003899336112734208$ и $B=100663296$ в формате плавающей точки одинарной точности равен $(1.49999982118606567382812 5-2^{-24})*2^{127}=255211734625884640294190107679323586560$, а после упаковки в двоичном и шестнадцатеричном представлении имеет вид:



S	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
			7				F				3				F				F				F				F				E

Рисунок 96

Как и прежде, можем сравнить результат, полученный ручным перемножением, с эталонной аппаратной реализацией на десктопе:

```
#include <iomanip>
#include <iostream>
#include <cstdint>

void print_float_as_hex(float* a) {
    uint32_t int_rep = *(uint32_t*)a;
    std::cout << std::hex << std::setfill('0') << std::setw(8) << int_rep << std::endl;
}

int main() {
    float a = 2535300898225003899336112734208.0f;
    float b = 100663296.0f;
    float c = a * b;

    std::cout << "a: " << std::fixed << std::setprecision(50) << a << "\n";
```

```
    std::cout << "b: " << std::fixed << std::setprecision(50) << b << "\n";
    std::cout << "c: " << std::fixed << std::setprecision(50) << c << "\n";

    print_float_as_hex(&c);

    return 0;
}
```

Наибольшее произведение

Возьмем два наибольших числа, представимых в одинарном float: $A=1.99999988079071044921875*2^{127}$ и $B=1.99999988079071044921875*2^{127}$.

Сумма истинных экспонент равна 254 или $254+127=381$ в коде со смещением. То есть, экспонента произведения не представима в формате одинарной точности.

Произведение мантисс даст значение с переполнением равное 3.9999995231628560077297152020037174224853515625. После нормализации вправо истинная мантисса произведения будет равна 1.99999976158142800386485760100185871124267578125. Экспонента после коррекции будет равна в коде со смещением $(254+1)+127=255+127=382$.

То есть, произведение максимальных значений в пределах одинарной точности даёт переполнение. Согласно стандарту IEEE 754 в таком случае устанавливается спец. значение inf.

Наименьшее произведение

Возьмем два наименьших числа, представимых в одинарном float: $A=1.0*2^{-126}$ и $B=1.0*2^{-126}$ (мы рассматриваем только нормализованные числа, с поддержкой денормализованных наименьшее представимое число равно $2^{-23}*2^{-126}=2^{-149}$).

Умножение даст $1*2^{(-126-126=-252)}$. Т.е. экспонента произведения в коде со смещением будет равна $-126-126+127=-252+127=-125$, что даёт ситуацию антипереполнения. То есть, истинное произведение настолько мало, что не хватает разрядности экспоненты для его представления. А поскольку мы не реализуем поддержку денормализованных чисел, то просто «падаем» в ноль, без перехода к денормализованному числу.

Но в данном примере даже после денормализации мы получим антипереполнение и переход к нулю, поскольку когда мы денормализуем мантиссу вправо, доведя экспоненту до представимого значения, равного -126, единственный значащий разряд мантиссы окажется за пределами разрядной сетки одинарной точности и в результате округления мы получим ноль.

Реализация деления — FDIV

В общем случае деление выполняется так:

$$\frac{MANT(A)*2^a}{MANT(B)*2^b}=\frac{MANT(A)}{MANT(B)}*\frac{2^a}{2^b}=\frac{MANT(A)}{MANT(B)}*2^{(a-b)}. \qquad (19)$$

То есть, вычисляем частное мантисс и разность экспонент.

А поскольку деление дробных нормализованных мантисс будем выполнять по схеме с неподвижным делителем без восстановления остатка, то как и при целочисленном делении возникает вопрос о необходимости и достаточности расширения разрядной сетки влево на один разряд для обеспечения безопасного сдвига влево как положительного остатка так и доп. кода отрицательного, и для корректного определения знака нового остатка.

В действительности, поскольку дробные мантиссы можно интерпретировать как целые числа или даже рассматривать чисто механически битовые паттерны в старшей части, все рассуждения и выводы касаемо расширения сетки влево, сделанные для целочисленного деления, остаются справедливыми и в случае деления дробных мантисс. Разница лишь в абсолютных значениях операндов и максимальном по модулю отрицательном остатке, который может быть получен на первой итерации: поскольку мантиссы нормализованы, то любой операнд на первой итерации не может быть меньше 2^{23} в целочисленной интерпретации. Но на выводы о достаточности расширения сетки влево эти различия не влияют поскольку различия только в абсолютных значениях и масштабе величин, а не в их взаимном отношении.

TODO: Подчеркнуть, что частное лежит в $[0.5, 1.99999988079071044921875]$ и поэтому при вычислении нет необходимости вычислять цифру частного с весом больше 2^0.

TODO: Нужен ли guard-разряд слева при схеме без восстановления остатка? С восстановлением нужен, это понятно, т. к. вполне может быть ситуация, когда MSB совмещены, но мантисса делимого/остатка меньше делителя и мы снова сдвигаем влево. UPD: Да, ниже мы показали, что без расширения сетки влево не обойтись, а также мы показали, что расширения на 1 разряд достаточно (как для положительного сдвигаемого остатка, так и для отрицательного). UPD: Мы потом показали, что в случае схемы с восстановлением остатка (то есть, в случае сдвига только положительного остатка влево) guard-разряд слева не обязателен, можно построить реализацию без него. Для схемы без восстановления, в случае сдвига как положительного, так и отрицательного остатка, мы, скорее всего, тоже можем придумать схему без расширения, но в итоге мы пришли к выводу, что расширение еще на один разряд влево упростит реализацию. Ну а про достаточность расширения мы выше все уже сказали.

Наименьшее частное мантисс

Возьмем два числа, наименьшее делимое и наибольший делитель: $A=1$ и $B=(2^{24}-1)*2^{-23}=1.99999988079071044921875$. Истинное частное будет равно $\frac{1}{1.99999988079071044921875}=$ 0.5[000000298023241640522577793751823529709787947522875518970222411765003905594581699048381987117647356846771052287283676104764706180376182816993166029045941176768611476934640224852575352941474493829875816695440810647059121552653405228460146693], где в квадратных скобках выделен период.

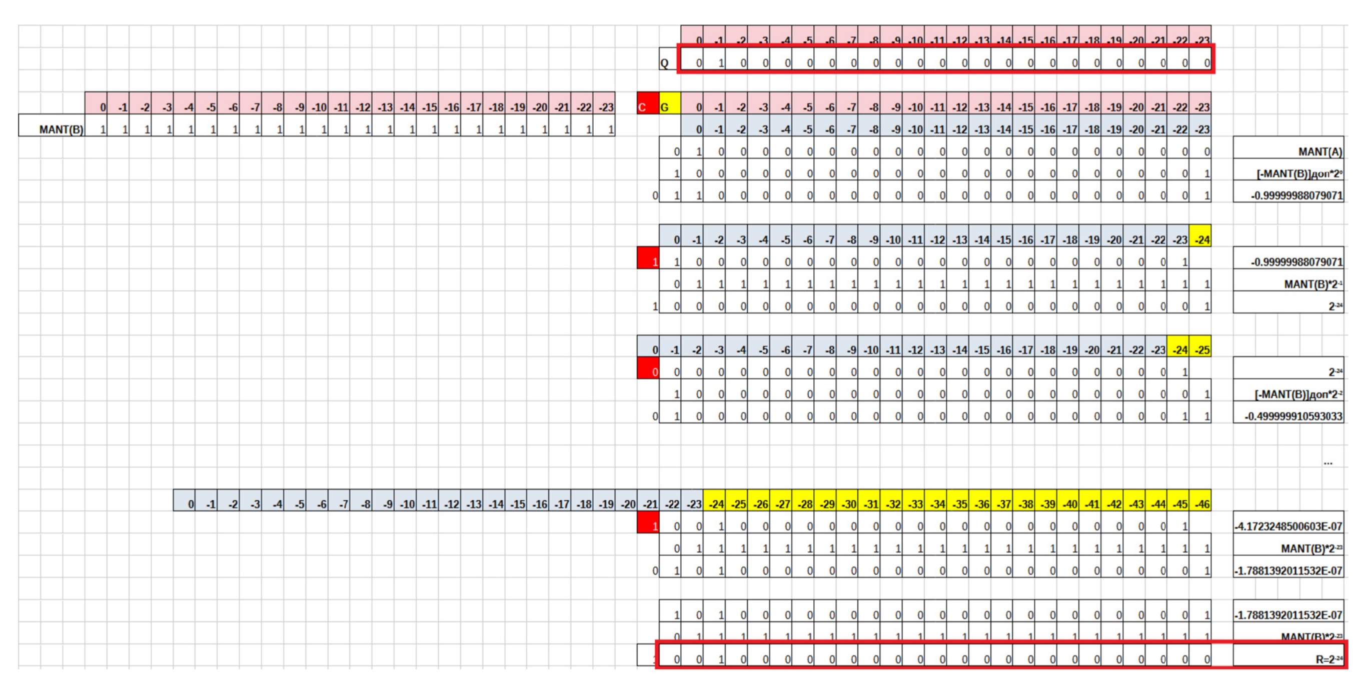


Рисунок 97 (разряды залитые красным — бит переноса)

Мы вычислили 24 цифры истинного частного, при этом последний положительный остаток равен 2^{-24} (относительно подвижной виртуальной сетки). Это означает, что за пределами сетки справа существуют ненулевые двоичные разряды. Но нам недостаточно разрядов частного для корректного округления. Поэтому мы должны вычислить частное с точностью до R-бита. Но истинное частное может быть (и так оно и есть в нашем примере) денормализовано вправо на один разряд, а это значит, что потребуется нормализация влево также на один разряд, из-за чего ненулевой бит частного в позиции R будет «вдвинут» в финальную разрядную сетку результата и будет потеряна информация для корректного округления. Поэтому нам необходимо вычислить частное с точностью до RG-битов, чтобы ненулевой истинный бит в позиции guard-разряда при нормализации влево встал нам место «потерянного» R-бита. И, наконец, если остаток окажется ненулевым, мы установим sticky-бит. В итоге, после нормализации влево, у нас будет истинное частное с точностью до RS-битов , чего достаточно для корректного округления. Если же

нормализация не требуется, у нас будет истинное частное с избыточной точностью до RGS-битов.

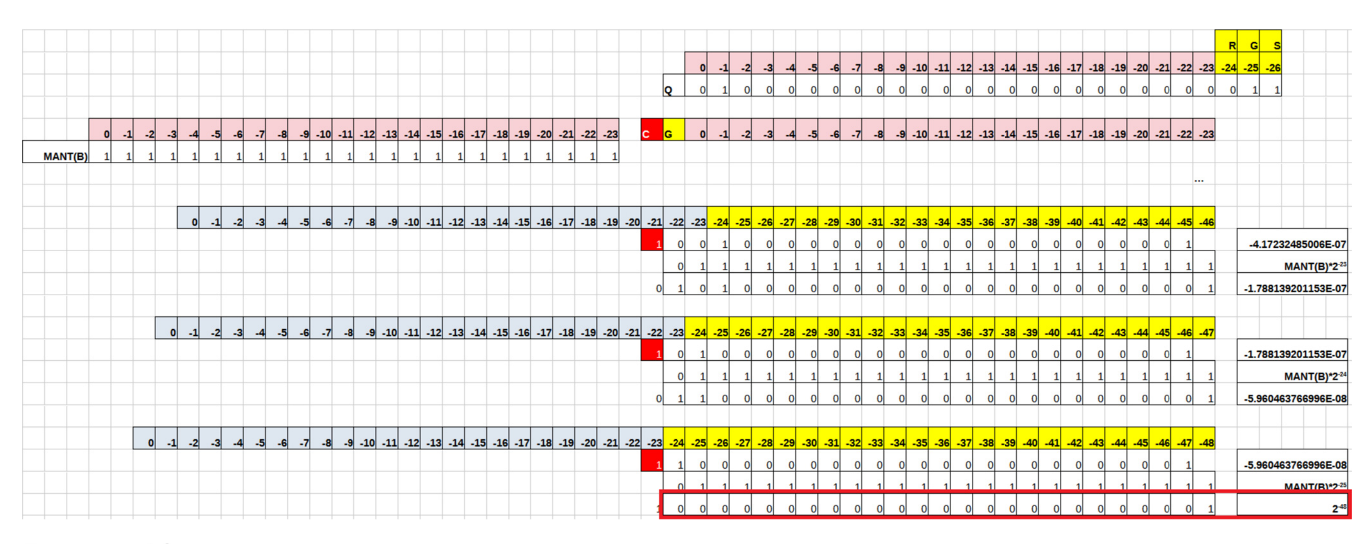


Рисунок 98

Мантисса частного вычислена. Экспоненты операндов нулевые, поэтому экспонента частного также равна нулю. Нормализуем мантиссу частного влево и округляем к ближайшему. Затем корректируем экспоненту, получая $(0-1)+127=126$ в коде со смещением. Наконец, пакуем согласно формату IEEE 754 и получаем финальное представление частного.

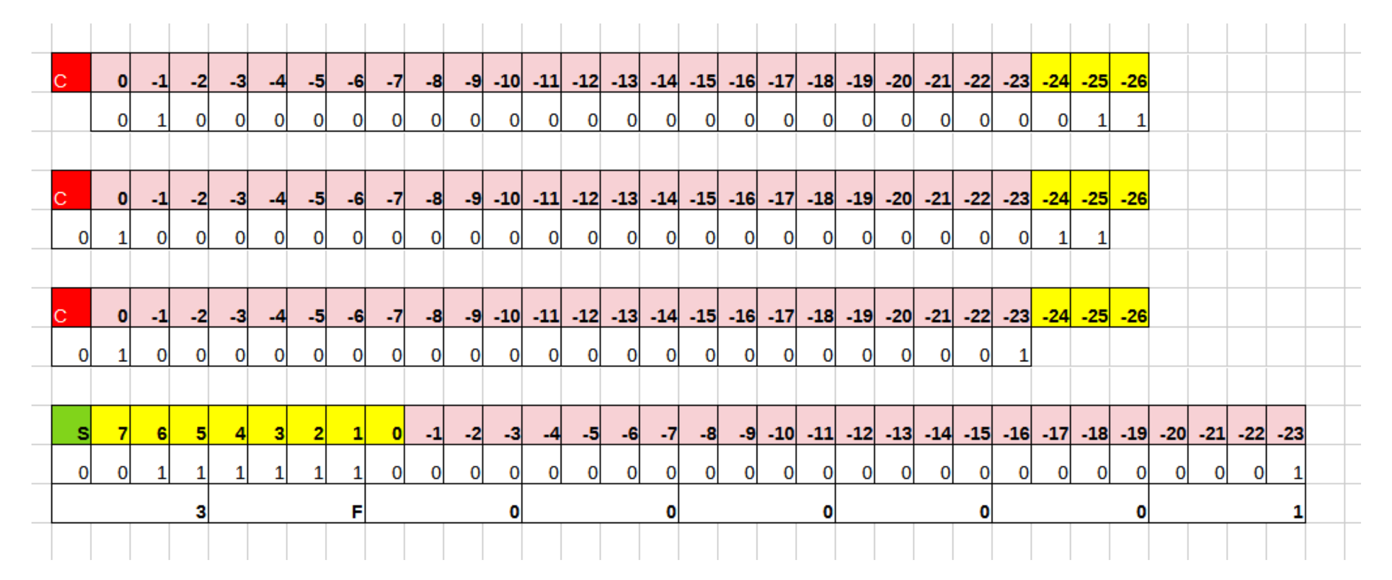


Рисунок 99

Для убедительности можно свериться с результатом на десктопе:

```
#include <iomanip>
#include <iostream>
#include <math.h>
#include <cstdint>

void print_float_as_hex(float* a) {
    uint32_t int_rep = *(uint32_t*)a;
    std::cout << std::hex << std::setfill('0') << std::setw(8) << int_rep << std::endl;
}

int main() {
    float a = 1.0f;
    float b = 1.99999988079071044921875f;
    float c = a / b;

    std::cout << "fixed: " << std::fixed << std::setprecision(150) << a << "\n";
    std::cout << "fixed: " << std::fixed << std::setprecision(150) << b << "\n";
    std::cout << "fixed: " << std::fixed << std::setprecision(150) << c << "\n";

    print_float_as_hex(&c);

    return 0;
}
```

NOTE: Частное может быть либо конечной либо бесконечной периодической дробью (делимое и делитель — это рациональные числа, доказательство — в теории чисел). Мы вычисляем частное с точностью до RGS, поэтому может возникнуть резонный вопрос: могут ли нормализованные делимое и делитель, умещающиеся в 24 разрядах (в случае одинарной точности) дать частное, которое будет конечной дробью с ненулевыми разрядами в пределах или за пределами RGS-зоны? Проще говоря, не будет ли так, что истинное частное — это конечная дробь, а мы при этом вычислили не все разряды, т. к. не хватило расширения сетки вправо?

Рассмотрим такой случай: допустим, что истинное частное содержит единичный разряд уже в позиции -24. Тогда, поскольку $Q*B=A$, сумма частичных произведений делителя с весами ненулевых разрядов частного должна дать исходное значение делимого A, у которого все разряды справа от разряда с индексом -23 нулевые.

Последним членом суммы частичных произведений будет делитель сдвинутый на 24 разряда вправо, а поскольку делитель не меньше единицы, то сумма частичных произведений, равная делимому A неизбежно будет содержать как минимум один ненулевой разряд справа от разряда с индексом -23, что противоречит начальным условиям относительно делимого A.

Таким образом, если частное — конечная дробь, то все ненулевые разряды вмещаются в сетку размера 24.

Наибольшее частное мантисс

Наибольшее частное равно $\frac{(2^{24}-1)*2^{-23}}{1}=1.99999988079071044921875$, т. е. — максимальному значению операнда.

Округление halfway

Напомним, что частное может быть либо нормализованным либо денормализованным вправо на один разряд. В обоих случая ситуация симметричного округления невозможна.

Возьмем нормализованное частное. Наименьшее его значение, которое даст ситуацию симметричного округления равно $1+2^{-24}=1.000000059604644775390625$. Хотя это значение и лежит в интервале $[0.5, 1.99999988079071044921875]$, оно является конечной дробью, которая занимает 25 разрядов, то есть, содержит ненулевой разряд с весом 2^{-24}, а выше мы показали, что при нормализованных делителе и делимом длиной в 24 разряда, такое частное получено быть не может, потому что иначе произведение делителя и такого частного неизбежно даст делимое, у которого в разрядной сетке за пределами индекса -23 будет как минимум один ненулевой разряд.

Теперь возьмем денормализованное частное. Наименьшее его значение, которое после нормализации даст ситуацию симметричного округления равно $0.5+2^{-25}=0.5000000298023223876953125$. И снова, по тем же соображения, что и выше, такое частное получено быть не может.

Таким образом, при реализации деления нет необходимости реализовывать проверку на halfway-округление.

Переполнение при округлении

Наименьшее нормализованное частное, которое после округления даст переполнение равно $(2^{24}-1)*2^{-23}+2^{-24}=1.999999940395355224609375$. Кроме того, что это конечная дробь из 25 разрядов (выше мы разбирали, почему такой результат невозможен), это значение больше максимального возможного частного.

Теперь разберем ситуацию с денормализованными значениями частного. Построим декартово произведение всех значений делимого и делителя:

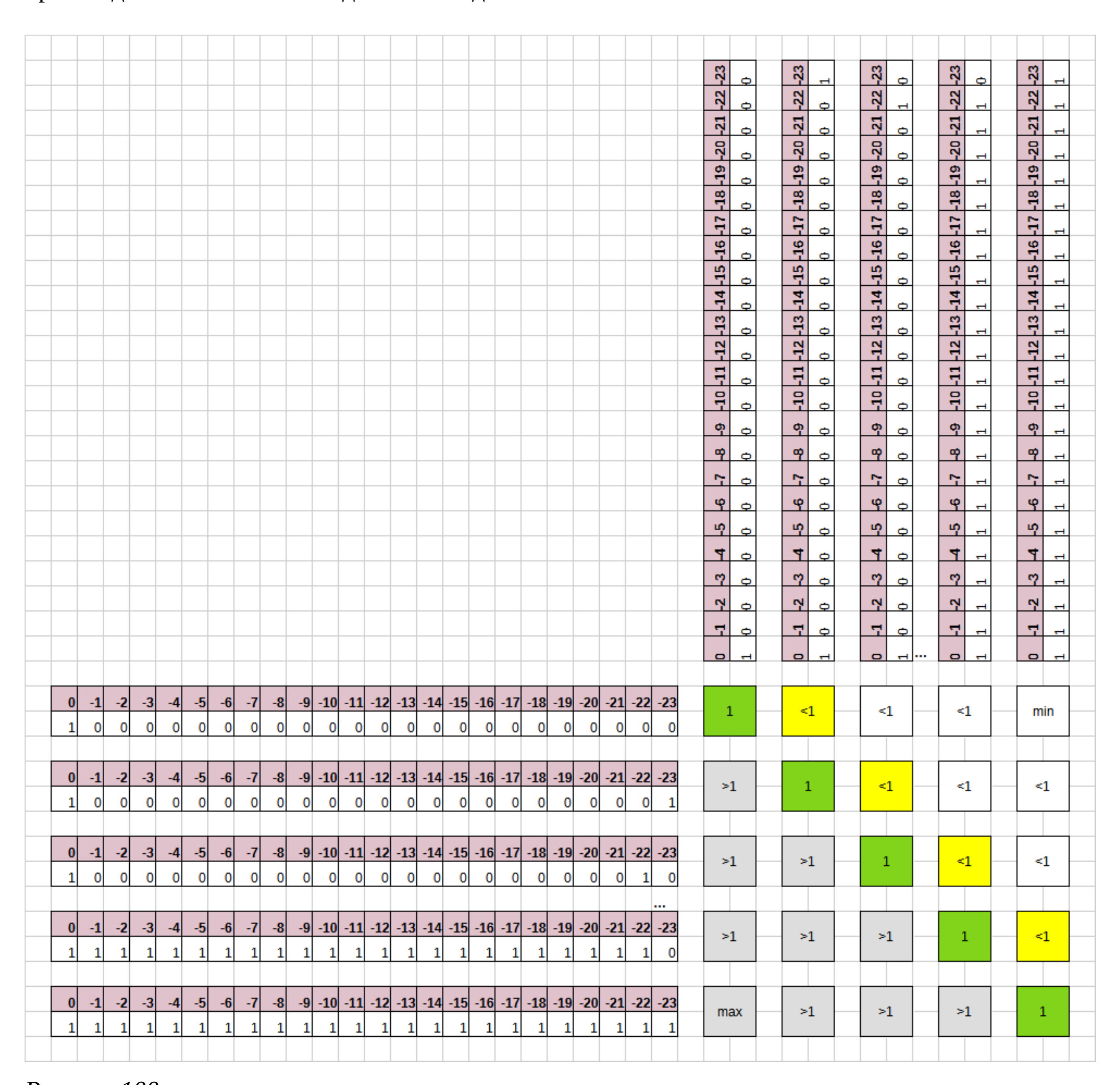


Рисунок 100

Видим, что все частные меньше единицы (т. е. денормализованные вправо на один разряд) находятся исключительно над диагональю.

Еще мы видим, что наибольшие значения денормализованных частных находятся сразу над диагональю.

Выразим теперь все денормализованные значения, которые находятся над диагональю (выделены желтым):

$$f(x)=\frac{x}{x+2^{-23}}. \tag{20}$$

Если мы вычислим самое верхнее частное из желтой диагонали и самое нижнее, то нижнее окажется больше верхнего. Таким образом, у нас есть гипотеза о том, что вниз по желтой диагонали денормализованные частные растут. Проверим её, вычислив предел выражения 20 (от неопределенности избавимся разделив числитель и знаменатель на x):

$$\lim_{x\to\infty}\frac{x}{x+2^{-23}}=\frac{1}{1+\frac{2^{-23}}{x}}=\frac{1}{1+0}=1. \tag{21}$$

Действительно, гипотеза подтвердилась, а это значит, что наибольшее денормализованное частное находится в самом низу диагонали и равно $\frac{(2^{24}-1)*2^{-23}-2^{-23}}{(2^{24}-1)*2^{-23}}$.

Вычислим это частное с точностью до RGS:

0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	G	S
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1

Рисунок 101

Очевидно, что после нормализации и округления переполнения не будет.

Таким образом, при реализации деления выполнять проверку на переполнение после округления нет необходимости.

Реализация алгебраического сложения — FADD

Одинаковые знаки

Возьмем положительное число $A=1.99993622303009033203125*2^{-126}$ и $B=1.99999988079071044921875*2^{-117}$.

Их мантиссы нормализованы, но порядки при этом не равны. Разность порядков составляет $-117-(-126)=-117+126=9$. То есть, прежде чем выполнять сложение, нам нужно расположить мантиссы друг под другом так, как они будут расположены в неограниченной разрядной сетке после умножения на свои характеристики – 2^p.

В самом наивном сценарии, мы могли бы вовсе избавиться от экспоненциальной записи, сдвинув мантиссу числа A вправо на 126 разрядов и мантиссу числа B – вправо на 117 разрядов. Но на уровне реализации мы не можем позволить себе разрядную сетку размером в 126 разрядов справа (более того, если порядок будет положительным, нам может потребоваться сдвиг влево на 127 разрядов). К тому же, нас интересует только разность порядков, которая составляет всего 9 разрядов.

Для экономии разрядной сетки мы можем оставить мантиссу одного из операндов неподвижной, а мантиссу другого операнда сдвинуть на разность порядков, тем самым получив выравнивание порядков операндов.

В нашем примере важно заметить, что если мы оставим неподвижной мантиссу первого операнда, порядок которого равен -126, то нам придется сдвинуть мантиссу второго операнда на 9 разрядов влево, но мы не хотим расширять разрядную сетку влево на 9 разрядов (к тому же, разность порядков может быть гораздо больше 9).

Но если мы сделаем предварительный своп операндов по такому принципу, что первым операндом всегда будет наибольший по модулю, то получим:

$A=1.99999988079071044921875*2^{-117}$ и $B=1.99993622303009033203125*2^{-126}$.

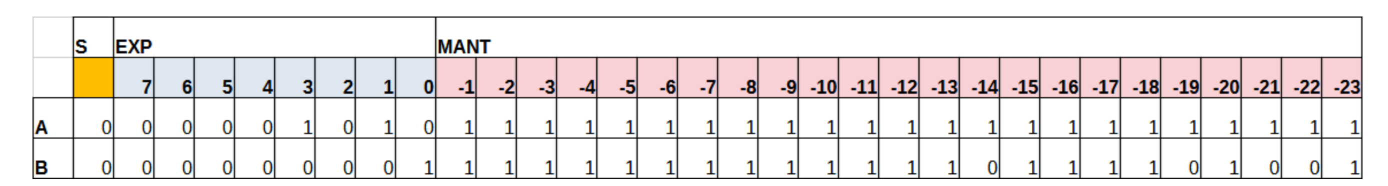
Таким образом, после свопа, для выравнивания порядков всегда достаточно сдвинуть второй операнд вправо на разность порядков.

Так выглядят операнды, поступившие на вход:

	S	EXP								MANT																						
		7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
A	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0	0	1
B	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Рисунок 102

Делаем своп так, чтобы первым операндом стал наибольший помодулю (перед свопом берем модули целочисленных представлений, образованных мантиссой и экспонентой в коде со смещением. Такое сравнение операндов в формате плавающей точки мы подробно разобрали в разделе Экспонента в коде со смещением.).



	S	EXP								MANT																						
		7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
A	0	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0	0	1

Рисунок 103

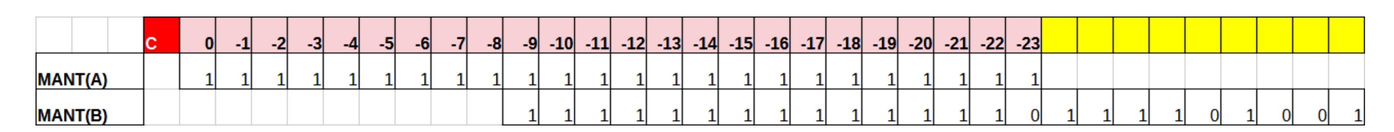
Теперь *A* содержит наибольший (по модулю) операнд, соответственно, теперь порядок *A* не меньше порядка *B*, а значит при вычислении разности порядков мы всегда получим неотрицательное значение.

Распаковываем мантиссы:

	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
MANT(A)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
MANT(B)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0	0	1

Рисунок 104

Разность порядков, как мы уже выяснили выше, равна 9. Операнд *A* имеет наибольший порядок, поэтому для выравнивания нам нужно сдвинуть мантиссу *B* вправо на разность порядков:



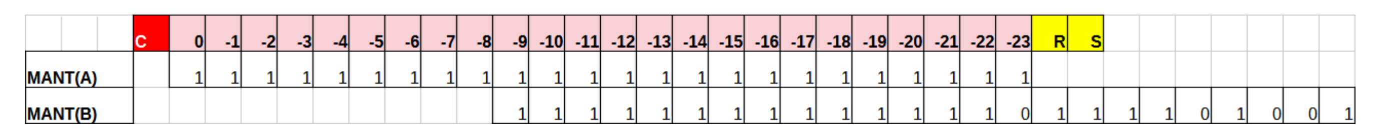
	C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23									
MANT(A)		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1									
MANT(B)											1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0	0	1

Рисунок 105

Здесь важно вспомнить, что разрядная сетка результата ограничена 24 разрядами и нам придется округлить результат сложения, при этом округление должно быть корректным, т. е. к ближайшему в соответствии с дефолтной схемой.

Также, помимо округления, нам еще важно учесть переполнение при сложении, которое в нашем случае будет.

При этом, при сложении операндов с одинаковыми знаками (как положительными так и отрицательными) денормализация вправо невозможна, поэтому для корректного округления нам достаточно иметь один R-бит и один S-бит.

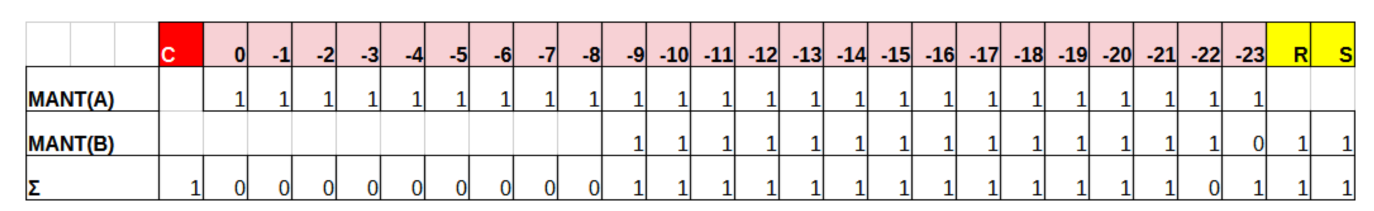


	C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	S							
MANT(A)		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1									
MANT(B)											1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0	0	1

Рисунок 106

В данном примере за пределами расширенной сетки есть ненулевые разряды, поэтому S-бит должен быть установлен в единицу. То есть, при выравнивании порядков, каждый раз, когда вправо сдвинут ненулевой бит, S-разряд устанавливается в единицу.

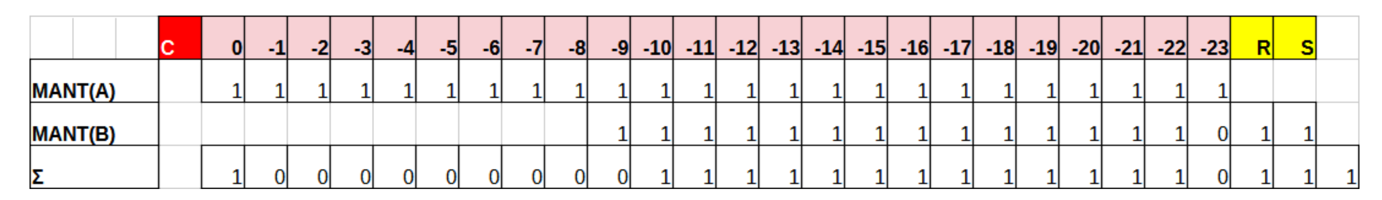
Теперь, после свопа и выравнивания порядков (порядок B теперь равен порядку A) мы можем выполнить сложение. При этом, поскольку мы выполнили округление мантиссы B до RS, после сложения мы получим не истинную сумму, а сумму округленную также до RS.



	C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	S
MANT(A)		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		
MANT(B)											1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
Σ	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1

Рисунок 107

Мы получили переполнение, поэтому нормализуем вправо и корректируем экспоненту суммы: $-117+1=-116$, или $-116+127=11$ в коде со смещением.



	C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	S	
MANT(A)		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1			
MANT(B)											1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	
Σ		1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1

Рисунок 108

И снова помним об ограниченной сетке, и в случае потери единичного бита справа устанавливаем S-бит в единицу (в нашем случае истинный бит в позиции S мантиссы суммы уже равен единице, поэтому повторная его установка не дает видимых изменений).

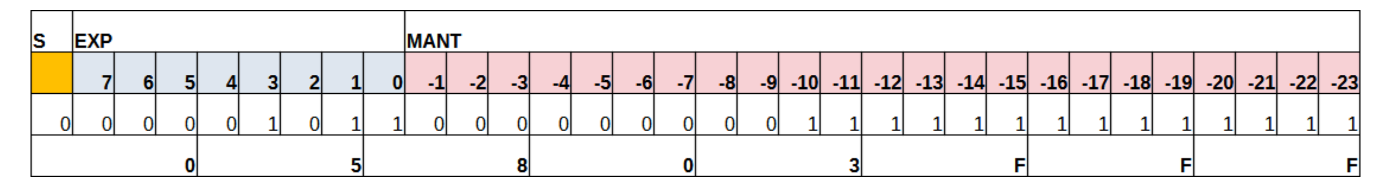
Округляем результат (в данном случае — в большую сторону).

	C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	S
Σ		1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
ROUND(Σ)		1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1		

Рисунок 109

NOTE: После округления, перед которым было переполнение и нормализация вправо, повторное переполнение невозможно. Действительно, поскольку было переполнение при сложение, значит возник бит переноса. Но «источником» бита переноса могут быть только разряды, индексы которых лежат в отрезке $[0,-23]$. При этом, у суммы, разряд, являющийся источником бита переноса, будет нулевым. Таким образом, после нормализации вправо, как минимум один разряд, лежащий в отрезке $[-1,-24]$ будет нулевым. Если нулевым окажется разряд с весом 2^{-24}, то это даст нулевой R-бит, а значит будет округление в меньшую сторону и переполнения не будет по определению. Если же нулевым будет разряд из отрезка $[-1,-23]$, то даже если отбрасывание RS-битов и прибавление бита с весом 2^{-23} даст бит переноса из позиции -23, этот бит переноса неизбежно «упадет» в нулевой бит. Но если переполнения при сложении не было, то после округления возможно переполнение, поэтому, в общем случае, на уровне реализации после округления требуется проверка на переполнение.

И, наконец, с учетом округления финальной суммы и коррекции экспоненты, пакуем результат в соответствии с IEEE 754 (знаки в данном случае одинаковые и положительные, поэтому знак суммы также положительный, поэтому просто берем знак первого аргумента: далее, при рассмотрении сложения с разными знаками, мы также будем брать знак первого аргумента, но аргументация будет немного другая).



S	EXP								MANT																						
	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
			0				5				8				0				3				F				F				F

Рисунок 110

Как обычно, сверяемся с аппаратной реализацией на десктопе:

```
#include <iomanip>
#include <iostream>
#include <math.h>
#include <cstdint>

void print_float_as_hex(float* a) {
    uint32_t int_rep = *(uint32_t*)a;
    std::cout << std::hex << std::setfill('0') << std::setw(8) << int_rep << std::endl;
}

int main() {
    float a = 1.99993622303009033203125f * powl(2, -126);
    float b = 1.99999880790710449218750f * powl(2, -117);
    float c = a + b;

    std::cout << "a: " << std::fixed << std::setprecision(150) << a << "\n";
    std::cout << "b: " << std::fixed << std::setprecision(150) << b << "\n";
    std::cout << "c: " << std::fixed << std::setprecision(150) << c << "\n";

    print_float_as_hex(&c);

    return 0;
}
```

Разные знаки

Возьмем два числа: $A = 1.99999892711639404296875 * 2^{-4}$ и $B = -1$.

	S	EXP								MANT																						
		7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
A	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
B	1	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

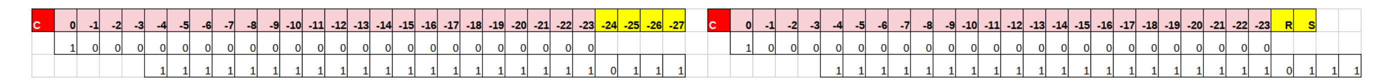
Рисунок 111

После взятия модулей чисел, также, как и в предыдущем разделе, выполняем своп, размещая наибольший (по модулю) операнд первым: $A = -1$ и $B = 1.99999892711639404296875 * 2^{-4}$. Напомним, что это нужно для реализации выравнивания порядков через сдвиг второго операнда вправо.

Но в ситуации разных знаков, своп обеспечивает не только выравнивание порядков, но и позволяет однозначно вычислять знак результата: если знаки одинаковые, как было в предыдущем разделе, то любой операнд, в т.ч. первый, содержит знак истинного результата; если же знаки разные, то мы вычисляем разность модулей, а знак разности будет равен знаку наибольшего по модулю операнда. То есть, независимо от сочетания знаков операндов, чтобы определить знак результата, мы берем знак первого операнда. Единственный сценарий, когда первый операнд не будет содержать знак истинного результата — когда разность равна нулю, то есть, когда модули операндов в точности совпадают. Но на уровне реализации это будет частный случай, для которого знак результата формируется отдельно.

Вычисление разности будем рассматривать параллельно: с расширением сетки на требуемое количество разрядов вправо (то есть, без потери точности) и только на два разряда (до RS-битов).

Выполняем распаковку мантисс и выравнивание порядков (разность порядков равна 4, поэтому сдвигаем второй операнд вправо на 4 разряда).



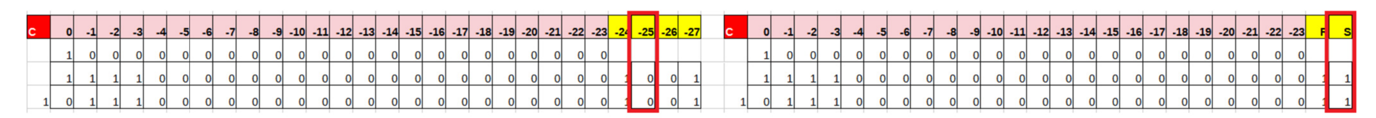
C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
					1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1

C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	S		
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
					1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1

Рисунок 112

Во втором случае (справа), поскольку теряются значащие разряды, округляем второй операнд до S-бита в большую сторону (или, что то же самое, просто устанавливаем S-бит в единицу, как признак того, что за пределами сетки есть ненулевые разряды).

Далее, поскольку знаки разные, вычисляем доп. код второго операнда и выполняем сложение.



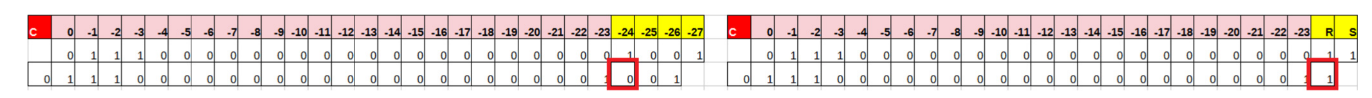
C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1

C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	S
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Рисунок 113

Сразу заметим, что в примере справа, после округления и вычисления доп. кода, бит в позиции -25 (S-бит) содержит не истинное значение: истинное значение в этой позиции равно нулю.

Кроме этого, мы получили денормализацию вправо на один разряд. Нормализуем влево и корректируем экспоненту (на рисунке не показана).



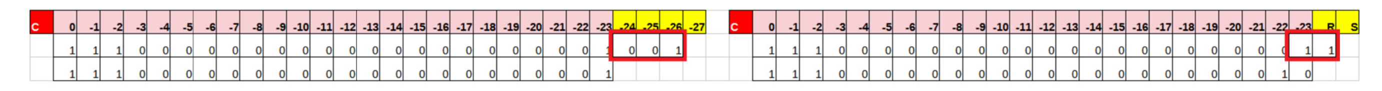
C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27
	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	

C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	S
	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	

Рисунок 114

Снова подмечаем, что после нормализации, в примере справа, бит в позиции R содержит не истинное значение.

Выполняем округление.



C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	-24	-25	-26	-27
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1				

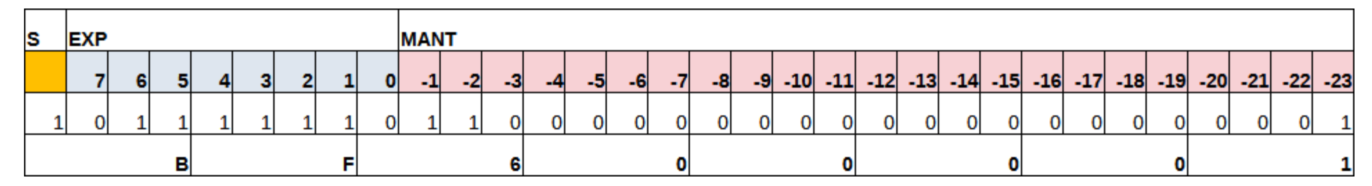
C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	S
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	
	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0		

Рисунок 115

Слева видим, что для округления истинного значения разности к ближайшему, нужно отбросить 2^{-26}. Справа же, где R-бит содержит не истинное значение, мы получили ситуацию симметричного округления, а поскольку бит в позиции -23 не равен нулю, то согласно дефолтной схеме округления (к ближайшему/к четному) мы должны прибавить 2^{-24}, получая некорректный результат округления, дающий большую ошибку.

Таким образом, мы видим, что, поскольку вычисление разности может привести к денормализации результата вправо (как в случае деления), нам необходимо расширить разрядную сетку вправо до RGS, чтобы после нормализации влево, в позицию R встало истинное значение, содержащееся в G-разряде, которое позволит корректно определить направление округления к ближайшему значению.

После упаковки, с учетом коррекции экспоненты после нормализации, и с учетом знака наибольшего по модулю операнда, получаем финальный результат:



S	EXP								MANT																						
	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
1	0	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
				B				F			6				0				0				0				0				1

Рисунок 116

Как и раньше, убеждаемся в корректности результата, сравнивая с эталонной аппаратной реализацией:

```
#include <iomanip>
#include <iostream>
#include <math.h>
#include <cstdint>

void print_float_as_hex(float* a) {
    uint32_t int_rep = *(uint32_t*)a;
    std::cout << std::hex << std::setfill('0') << std::setw(8) << int_rep << std::endl;
}

int main() {
    float a = 1.99999892711639404296875f * powl(2, -4);
    float b = -1.0f;
    float c = a + b;

    std::cout << "fixed: " << std::fixed << std::setprecision(150) << a << "\n";
    std::cout << "fixed: " << std::fixed << std::setprecision(150) << b << "\n";
    std::cout << "fixed: " << std::fixed << std::setprecision(150) << c << "\n";

    print_float_as_hex(&c);

    return 0;
}
```

NOTE: Поскольку всегда выполняется своп модулей операндов, разность модулей никогда не будет отрицательной, то есть, до определения знака результата по знаку наибольшего по модулю операнда, разность всегда будет неотрицательной. Более того, поскольку выполняется вычитание, переполнение при вычислении разности, конечно, невозможно.

Денормализация разности больше, чем на один разряд вправо

При вычислении разности возможна денормализация больше, чем на один разряд вправо.

Возьмем числа $A=1$ и $B=-1.99999988079071044921875*2^{-1}$.

	S	EXP								MANT																						
		7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
A	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Рисунок 117

Своп не требуется, выравниваем порядки сдвигая B вправо на 1 разряд.

C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	G	S
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		

Рисунок 118

Выполняем вычитание и получаем 2^{-24}.

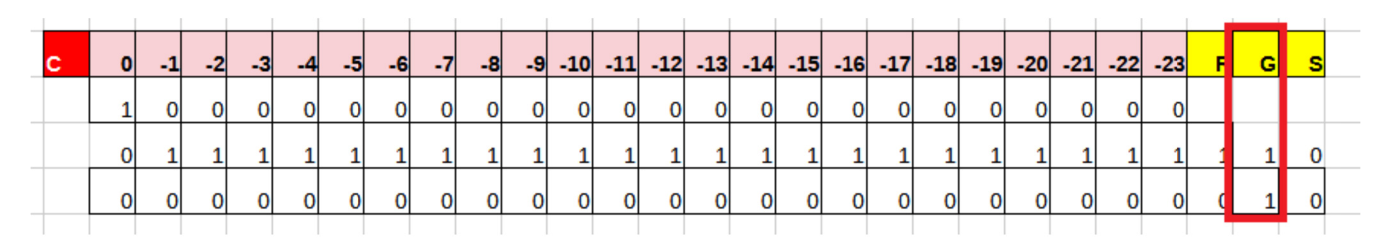
C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	G	S		
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0					A
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0		B
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0		A-B

Рисунок 119

Из рисунка хорошо видно, что если разность получается денормализованной, то операнд B при этом не может быть денормализован вправо больше, чем на один разряд, поскольку сумма разности $A - B$ и операнда B должна дать нормализованное значение A длиной в 24 разряда, а для этого нужно, чтобы бит переноса «упал» в позицию 0, что в свою очередь требует единичного бита у операнда B в позиции -1.

Более того, поскольку оба операнда имеют длину не больше 24 разрядов (имеется ввиду количество ненулевых разрядов), то наибольшая денормализация вправо составляет 24 разряда. Также, из ограничения на максимальную длину операндов следует, что если результат денормализован, то его последний ненулевой разряд находится в позиции не меньше -24.

Действительно, предположим, что можно получить денормализацию хотя бы на 25 разрядов вправо.



C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	G	S
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

Рисунок 120

Но тогда B должен иметь длину в 25 разрядов, а это невозможно при наших ограничениях.

Наконец, предположим, что при денормализации больше, чем на 1 разряд вправо, у разности может быть ненулевой разряд в позиции с индексом меньше, чем -24, например, хотя бы в позиции -25.

C	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	R	G	S
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0			
	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0

Рисунок 121

Снова, хотя денормализация меньше максимальной, из-за ненулевого разряда в позиции -25 операнд B должен иметь длину в 25 разрядов, что невозможно.

Таким образом, хотя при вычислении разности возможна денормализация больше, чем на 1 разряд вправо, тем не менее, все значащие биты такой денормализованной разности будут находиться в пределах расширенной сетки до S-бита, а значит будут содержать истинные значения, что означает, что при нормализации влево они будут безопасны для сдвига.

Проще говоря, если разность денормализована вправо более, чем на один разряд, она является истинной, а не округленной, а значит все её значащие разряды вмещаются в пределах расширенной сетки и нормализация влево безопасна в том смысле, что невозможна ситуация, когда в финальную разрядную сетку результата будут «вдвинуты» неистинные значения разрядов.

Поэтому расширения сетки до RGS достаточно и дополнительные guard-разряды не нужны.

Переполнение при округлении

Если разность получается нормализованной, то, довольно очевидно, что переполнения при округлении быть не может. Действительно, наименьшее нормализованное значение, дающее переполнение при округлении равно $(2^{24}-1)*2^{-23}+2^{-24}$, а это на 2^{-24} больше, чем наибольшее значение любого операнда, включая A.

Если же разность денормализована на один разряд вправо, то после нормализации при округлении возможно переполнение. Например, когда $A=1$ и $B=-2^{-25}$.

Таким образом, в общем случае, после выполнения нормализации и округления требуется проверка на переполнение.

Реализация вычитания — FSUB

Если второй операнд положительный, то знак меняется на отрицательный. Если второй операнд уже отрицательный, то знак меняется на положительный. Таким образом, на уровне реализации FSUB просто XOR’ит знак второго операнда и вызывает FADD.

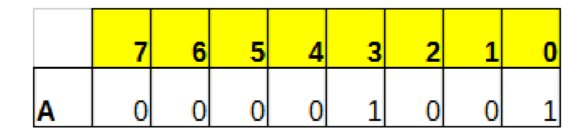
Конвертация

Подпрограммам конвертации atof и ftoa необходимы вспомогательные подпрограммы: itof и ftoi для преобразования целого числа в float и усечения float до целого числа.

Конвертация int в float — itof

Нам потребуется преобразовывать в формат float двоичные целые числа из интервала $[0,10)$, поэтому конвертация будет рассматриваться в пределах одного байта.

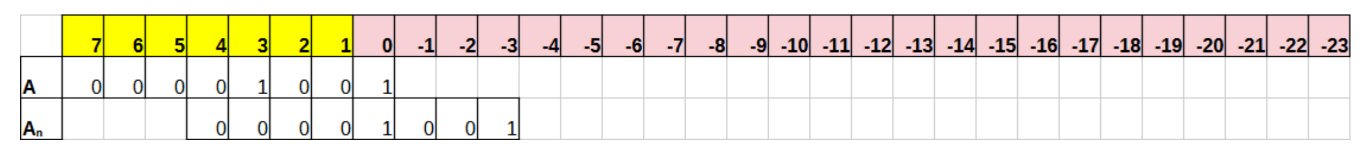
Возьмем для примера наибольшее целое число $A=9$.



	7	6	5	4	3	2	1	0
A	0	0	0	0	1	0	0	1

Рисунок 122

Первое, что нам нужно сделать, чтобы упаковать это число в float – нормализовать его вправо так, чтобы оно стало меньше 2:

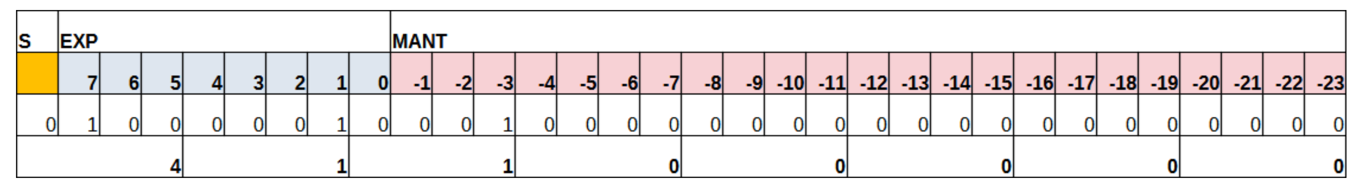


	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
A	0	0	0	0	1	0	0	1																							
A_n				0	0	0	0	1	0	0	1																				

Рисунок 123

Для нормализации потребовалось 3 сдвига вправо, т. е. — деление на $2^3=8$. Значит двоичная экспонента $EXP=3$ или $3+127=130$ в коде со смещением. Таким образом, исходное число $A=9$ в двоичном экспоненциальном представлении имеет вид $1.125*2^3$.

Наконец, после упаковки число $A=9$ принимает вид:



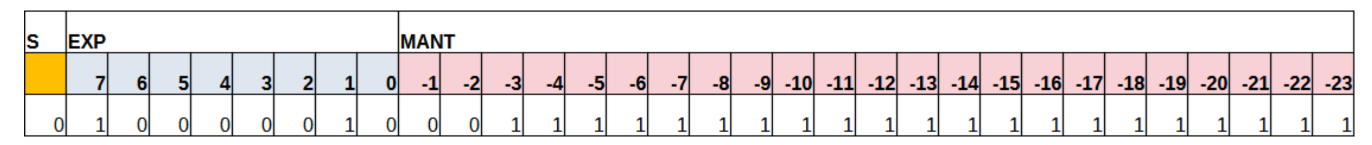
S	EXP								MANT																						
	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			4				1				1				0				0				0				0				0

Рисунок 124

Конвертация float в int — ftoi

Нам также понадобится конвертировать числа в формате float в целые, а точнее — усекать десятично нормализованные числа в формате плавающей точки до целой части (а это значит, что на выходе мы будем получать целые числа в пределах байта).

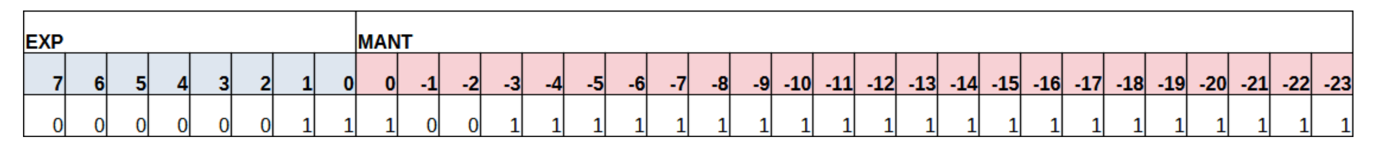
Возьмем наибольшее десятично нормализованное число, представимое в float одинарной точности: $A=10-(2^{-23}*2^{3})=9.99999904632568359375$.



S	EXP								MANT																						
	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
0	1	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Рисунок 125

Распаковываем экспоненту и мантиссу:



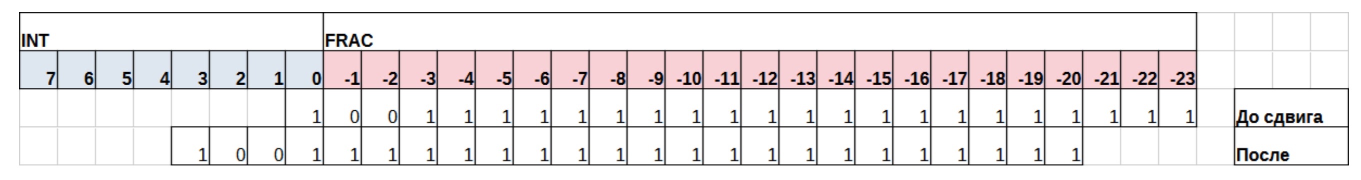
EXP								MANT																							
7	6	5	4	3	2	1	0	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23
0	0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Рисунок 126

Число десятично нормализовано, а это значит, что после двоичной денормализации (т. е. сдвига влево на величину экспоненты), в целой части двоичного представления будет значение в отрезке $[1,9]$, т. е. будет умещаться в пределах одного байта.

Еще одно довольно очевидное следствие того факта, что число десятично нормализовано — двоичная экспонента в коде без смещения лежит в отрезке $[0,3]$, что означает, что денормализация требуется только влево.

Денормализуем мантиссу влево на количество разрядов, равное двоичной экспоненте:



INT								FRAC																							
7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20	-21	-22	-23	
							1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	До сдвига
				1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1				После

Рисунок 127

Теперь целая часть INT размером в байт содержит двоичное представление целой части десятично нормализованного числа в формате float, которая равна 9.

Преобразование числовой ASCII-строки в число в формате плавающей точки — atof

Допустим, у нас есть числовая строка, представляющая дробное число $A = 123.45$. Строка эта оканчивается нулевым ASCII-символом NUL.

Эта числовая строка обозначает некоторое количество. Задача в том, чтобы выразить это же количество в формате бинарной плавающей точки, т. е. — в двоичном виде.

Будем читать строку начиная с самого старшего десятичного разряда, или, другими словами, с первого символа.

Каждая десятичная цифра в строке представлена ASCII-кодом в формате 0x3X, где X – шестнадцатеричное представление двоичного числа от 0 до 9, которое обозначается этим ASCII-кодом. Таким образом, чтобы извлечь число из цифры, достаточно отбросить старший полубайт ASCII-кода и мы получим двоичное число, выражаемое этой цифрой.

Допустим, мы считали первую цифру — 1 и извлекли число 1. Допустим также, что это единственная цифра в числовой строке, тогда для преобразования в float нам достаточно воспользоваться подпрограммой itof, которую мы разобрали выше. То есть, мы просто двоично нормализуем извлеченное число и пакуем в формат IEEE 754.

Далее, читаем следующий символ строки. Следующий символ оказывается равным 2. Также, извлекаем число 2 из ASCII-кода цифры и преобразуем в float. Поскольку в строке кроме единицы оказался еще один символ, то наше первое предположение о том, что считанная единица представляет разряд единиц, оказалось ложным. Поэтому наша следующая гипотеза в том, что единица представляет как минимум разряд десятков, а вот считанная двойка — разряд единиц. Умножаем текущий аккумулятор, который уже в формате float и содержит единицу, на 10. Аккумулятор теперь содержит число 10 в формате плавающей точки. Кроме этого, у нас есть число 2 в формате плавающей точки. Складываем их, используя арифметику с плавающей точкой, и получаем 12 в формате плавающей точки.

Читаем следующий символ — 3. Также, извлекаем число, пакуем в float с помощью itof. Снова, наше предположение о том, что единица представляет разряд десятков, а двойка — разряд единиц, оказалось ложным, а значит, единица представляет как минимум разряд сотен, а двойка — разряд десятков, ну а считанная только что тройка — разряд единиц. Умножаем аккумулятор на 10 и получаем число 120 в формате float. Складываем с тройкой и получаем 123, опять же, в формате float.

Следующий символ — это десятичная точка. Это значит, что мы сформировали целую часть числа в формате плавающей точки и далее пойдут дробные десятичные разряды.

NOTE: Чтобы алгортим оставался унифицированным, мы завышаем масштаб истинного числа так, чтобы дробная часть стала нулевой, то есть, в данном случае, умножаем истинное число на $10^2 = 100$ и получаем 12345 (это концептуальное описание, на уровне реализации мы будем умножать итеративно). При этом, мы запоминаем, что масштаб завышен и для возврата к истинному десятичному порядку нужно умножить число 12345 на 10^{-2}.

Пропускаем символ точки и переходим к формированию дробных десятичных разрядов. Считываем следующую цифру — 4, извлекаем число 4, преобразуем в float. Мы условились завысить масштаб, чтобы интерпретировать четверку, как разряд единиц, поэтому домножаем целую часть истинного числа в аккумуляторе на 10, получая 1230 и прибавляем число 4, получая 1234. При этом запоминаем, что мы завысили масштаб на 1 десятичный порядок: $OVERSCALE=1$, $OVERSCALE=OVERSCALE*10$.

Считываем следующую дробную цифру — 5, извлекаем число 5 и преобразуем в float. Домножаем аккумулятор на 10, получая 12340 и прибавляем 5, что дает 12345. Порядок истинного числа теперь завышен на два десятичных порядка, фиксируем это: $OVERSCALE=OVERSCALE*10=1*10*10=10^2=100$.

Считываем последний символ — он оказывается концом строки NUL. Мы считали всю числовую строку и сформировали число в формате плавающей точки, но при этом завысили порядок в 100 раз, то есть — на величину $OVERSCALE$. Делим аккумулятор на $OVERSCALE=100$ и получаем истинное число 123.45.

NOTE: Для простоты изложения мы опустили тот факт, что после восстановления истинного порядка делением на 100 мы получим искажение истинного значения из-за того, что десятичное число 123.45 не представимо точно в двоичном виде и при делении 12345 на 100 в бесконечной разрядной сетке мы будем получать бесконечную периодическую двоичную дробь, являщуюся бесконечным приближением к 123.45. А поскольку фактическая разрядная сетка ограничена, мы после деления выполним округление к ближайшему числу, получив конечную двоичную дробь, которая, понятное дело, будет только конечным приближением к 123.45 и при обратной конвертации в десятичную строку (из бинарного float) не даст в точности десятичное число 123.45.

Для наглядности, ниже приведена программа, которая выводит значение числа 12345 в формате float, значение переменной *OVERSCALE* и финальное значение, после восстановления порядка:

```
#include <iomanip>
#include <iostream>
#include <math.h>
#include <cstdint>

void print_float_as_hex(float* a) {
    uint32_t int_rep = *(uint32_t*)a;
    std::cout << std::hex << std::setfill('0') << std::setw(8) << int_rep << std::endl;
}

int main() {
    float a = 12345.0f;
    float b = 100.0f;
    float c = a / b;

    std::cout << "a: " << std::fixed << std::setprecision(150) << a << "\n";
    std::cout << "b: " << std::fixed << std::setprecision(150) << b << "\n";
    std::cout << "c: " << std::fixed << std::setprecision(150) << c << "\n";

    print_float_as_hex(&c);

    return 0;
}
```

Преобразование числа в формате плавающей точки в ASCII-строку в экспоненциальном формате — ftoae

Результаты вычислений могут быть как очень маленькими, так и очень большими, а видимая область строки в LCD ограничена 16 символами, поэтому все результаты будем выводить в экспоненциальном формате с показом десятичной мантиссы и экспоненты, за исключением случая, когда число уже десятично нормализовано, то есть, когда десятичная экспонента равна нулю.

Десятичная нормализация числа в формате плавающей точки

Поскольку для любого числа в формате плавающей точки мы хотим получить числовую строку в десятичной экспоненциальной записи, нам необходимо число в формате float предварительно десятично нормализовать, запомнив при этом значение исходного десятичного порядка.

Всего возможно три случая:

1. Исходное число уже десятично нормализовано.

   Если исходное число уже нормализовано десятично, то оно лежит в отрезке $[1,10)$. То есть, в формате float его двоичный порядок лежит в отрезке $[0,3]$ или $[127,130]$ в коде со смещением.

2. Исходное число денормализовано влево.

   Если исходное число денормализовано влево, то оно лежит в отрезке $[10,(2^{24}-1)*2^{-23}*2^{127}]$, а его двоичный порядок лежит в $[3,127]$ или $[130,254]$ в коде со смещением.

3. Исходное число денормализовано вправо.

   Если же исходное число денормализовано вправо, то оно лежит в $[(2^{24}-1)*2^{-23}*2^{-126},1-(2^{-23}*2^{-1})=0.999999940395355224609375]$ а его двоичный порядок отрицательный и лежит в $[-126,-1]$ или $[1,126]$ в коде со смещением.

Таким образом, первое, что мы можем сделать — проверить двоичный порядок на отрицательное значение, и, если он отрицательный, то мы умножаем число на 10 до тех пор, пока порядок не перестанет быть отрицательным (не забывая при этом накапливать десятичную экспоненту, знак которой будет отрицательным).

NOTE: Если очередное умножение дало неотрицательную двоичную экспоненту, то такое значение десятично нормализовано. Допустим, что это не так, тогда существует такое денормализованное вправо значение, которое после умножения на 10 сразу даст десятичную денормализацию влево. Возьмем наибольшее денормализованное вправо значение равное $(2^{24}-1)*2^{-23}*2^{-1}=1-(2^{-23}*2^{-1})=0.999999940395355224609375$ и умножим его на 10, получив 9.99999904632568359375. Понятно, что если мы будем брать значения меньше граничного и также умножать на 10, то мы для них тем более не получим денормализацию влево.

Далее, если двоичный порядок неотрицательный, тогда число либо уже нормализовано либо денормализовано влево. Делим число на 10, пока порядок не станет отрицательным. Если число уже нормализовано, то первое же деление на 10 даст денормализацию вправо и т.о. — отрицательную двоичную экспоненту. Если же число денормализовано влево, то если мы получили на очередном шаге отрицательную двоичную экспоненту, это признак того, что предыдущее значение уже было нормализовано десятично. Значение десятичной экспоненты, понятно, будет иметь положительный знак.

NOTE: Если очередное деление на 10 дало денормализацию вправо, то предыдущее значение было нормализовано десятично. Допустим, что это не так, тогда существует десятично денормализованное влево значение, которое после деления на 10 дает сразу денормализацию вправо. Возьмем наименьшее десятично денормализованное значение равное десяти. Легко убедиться, что после деления на 10 мы получим десятично нормализованное значение равное единице. Понятно, что если теперь будем брать значения больше десяти и делить на 10, то тем более не получим денормализацию вправо, т. к. значение частного будет только расти.

Преобразование десятично нормализованного числа в строку — ftoan

На данном этапе число уже десятично нормализовано, а это значит, что для извлечения целой части достаточно усечь число до int с помощью уже разобранной нами подпрограммы FTOI принцип работы которой в том, что мы денормализуем число двоично, «раскрывая» двоичную экспоненту и просто «забираем» целую часть, в которой оказывается двоичное число в отрезке $[1,10)$. После извлечения целой части прибавляем 0x30 и получаем ASCII-код десятичной цифры, которая обозначает число в целой части.

Далее, нам необходимо убрать из исходного числа num только что извлеченную целую часть. Преобразуем целую часть к float с помощью ITOF и просто вычитаем из исходного числа, пользуясь уже разработанным вычитанием FSUB в формате плавающей точки. Понятно, что полученное значение будет меньше единицы.

Далее остается извлечь дробные десятичные разряды. Текущее значение меньше единицы, а значит двоичная экспонента в коде без смещения меньше нуля (это можно использовать как признак на уровне реализации). Умножаем текущее значение на 10:

- Если после умножения на 10 число все еще меньше единицы, то есть, двоичная экспонента все еще отрицательная, значит очередной десятичный дробный разряд равен нулю — просто устанавливаем ноль в формируемой строке и продолжаем умножение на 10.
- Если же после умножения на 10 число оказалось не меньше единицы, значит, как мы выше разбирали, оно десятично нормализовано. Снова усекаем до целого через FTOI, прибавляем к извлеченному целому значению 0x30, получая ASCII-код десятичной цифры для текущего дробного разряда, конвертируем целое в float через ITOF и вычитаем из текущего значения num, снова получая значение меньше единицы.

Так повторяем до тех пор, пока не извлечем заданное количество разрядов precision. Значение precision при этом зависит от того, сколько символов уже зарезервировано из доступных на LCD. А это определяется следующими факторами:

- Максимальная длина строки LCD-дисплея (16 в нашем случае).
- Наличие цифры целой части (всегда отображается, даже если равна нулю, поэтому всегда резервируем один символ).
- Наличие знака числа (резервируем символ только под отрицательный знак).
- Наличие десятичной точки (всегда резервируем один символ).
- Наличие десятичной экспоненты. Если она нулевая, то нет необходимости резервировать под неё место на LCD, иначе — резервируем под неё 4 знака: $\{+|-\}\{E\}\{9\}\{9\}$.

Преобразование десятичной экспоненты в строку

Напомним, что десятичный порядок — это счетчик, который показывает сколько раз мы разделили или умножили число на 10 при нормализации.

Наибольшее десятичное число, представимое в бинарном float одинарной точности равно $(2^{24}-1)*2^{-23}*2^{127}=3.4028234663852885981170418348451692544 0*10^{38}$.

Наименьшее десятичное число, представимое в бинарном float одинарной точности (без реализации денормализованных чисел) равно $2^{-126}\approx 1.1754943508*10^{-38}$.

Таким образом, после нормализации модуль десятичной экспоненты принимает значения в отрезке $[0,38]$.

Следовательно, для конвертации экспоненты в ASCII-строку достаточно разделить её нацело на 10. Неполное частное при этом будет содержать числовое значение разряда десятков десятичного представления экспоненты, которое находится в отрезке $[0,3]$. Остаток, который по определению меньше делителя, то есть меньше 10, будет содержать числовое значение разряда единиц того же десятичного представления экспоненты.

Таким образом, для конвертации достаточно целочисленного деления в пределах байта (то есть, в пределах одного 8-разрядного регистра). Более того, поскольку мы достоверно знаем, что частное не превышает 3 (0b00000011), то нет необходимости вычислять все разряды частного в пределах байта, достаточно только двух младших, а значит, достаточно только двух итераций деления вместо полных восьми.

Ввод операндов с клавиатуры

Преобразование сырых кодов клавиш в ASCII-коды

На клавиатуре 16 клавиш. Клавиатура подключена к энкодеру MM74C922. При нажатии клавиши энкодер устанавливает бит готовности данных и выставляет на порту 4-битный код нажатой клавиши. Коды меняются в отрезке [0x00,0x0F]. Далее по тексту мы предполагаем, что пины строк и столбцов клавиатуры подключены к энкодеру таким образом, что нажатие клавиш слева направо и снизу вверх дает коды в возрастающем порядке.

Каждый байт в SRAM имеет двухбайтовый адрес вида 0xHHLL, где HH – старший байт адреса, а LL – младший.

Мы выбираем любой адрес в SRAM так, чтобы самый младший полубайт был равен нулю. Например, возьмем адрес 0x3FA0. Далее, начиная с этого адреса мы записываем в SRAM ASCII-коды символов, которые хотим «привязать» к соответствующим клавишам клавиатуры.

Адрес (2 байта)				**ASCII-код (1 байт)**	**Символ**	**Функциональное назначение**
3	F	A	0	0x43	'C'	Сброс
3	F	A	1	0x30	'0'	Цифра 0
3	F	A	2	0x2E	'.'	Десятичная точка
3	F	A	3	0x2F	'/'	Оператор деления
3	F	A	4	0x37	'7'	Цифра 7
3	F	A	5	0x38	'8'	Цифра 8
3	F	A	6	0x39	'9'	Цифра 9
3	F	A	7	0x78	'x'	Оператор умножения
3	F	A	8	0x34	'4'	Цифра 4
3	F	A	9	0x35	'5'	Цифра 5
3	F	A	A	0x36	'6'	Цифра 6
3	F	A	B	0x2D	'-'	Оператор вычитания или знак минус
3	F	A	C	0x31	'1'	Цифра 1
3	F	A	D	0x32	'2'	Цифра 2
3	F	A	E	0x33	'3'	Цифра 3
3	F	A	F	0x2B	'+'	Оператор сложения

Таблица 1 (желтым выделены полубайты адреса, напрямую соответствующие кодам энкодера)

Теперь, при каждом нажатии клавиши мы берем базовый адрес ASCII-таблицы 0x3FA0 и подставляем полубайтовый код энкодера [0x00, 0x0F]в младший полубайт адреса ASCII-таблицы и по полученному адресу читаем ASCII-код, привязанный к нажатой клавише.

Допустим, что нажата клавиша с цифрой 5, это десятая клавиша, если считать слева направо и снизу вверх. Следовательно, энкодер выдаст полубайтовый код 0x09. Подставляем этот код в базовый адрес таблицы и получаем новый адрес 0x3FA[9], читаем байт по этому адресу и получаем 0x35, то есть — ASCII-код цифры 5.

То есть, сырой код энкодера используется как младший полубайт адреса ASCII-таблицы в SRAM. Нажимая клавиши, пользователь, по сути, адресуется напрямую к байтам в ОЗУ, которые хранят ASCII-коды.

Обеспечение корректного ввода

Мы не должны позволить пользователю ввести некорректную десятичную числовую строку. Например, мы не должны допустить, чтобы пользователь мог ввести что-то вроде «--0.5.12.-».

Основная идея в том, что сразу после включения питания калькулятора, мы воспринимаем нажатие определенных клавиш из белого списка. Затем, после того, как пользователь нажал некоторую клавишу из белого списка, мы начинаем воспринимать нажатие клавиш уже из другого белого списка и т. д.

Например, сразу после включения питания пользователь может начать ввод числа с символа минуса (если это отрицательное число) или с любой цифры, включая ноль (если это дробь меньше единицы). При этом понятно, что пользователь не может начать ввод числа с символа десятичной точки. То есть, для начального состояния калькулятора (назовем его S0) белый список имеет вид [-,0-9,C]. Понятно, что нажатие клавиши сброса C должно быть доступно всегда, независимо от текущего состояния.

Теперь допустим, что пользователь нажал минус, тогда доступные для нажатия клавиши меняют состав: [0-9,C]. То есть, пользователь не может ввести второй символ минуса подряд, но может ввести цифру или вовсе сбросить ввод.

Такой механизм ввода реализуется через конечный автомат. Мы определяем начальное состояние S0, определяем список доступных для нажатия клавиш в этом состоянии, а для каждой клавиши (или группы клавиш) определяем список состояний, в которые калькулятор должен перейти при нажатии соответствующей клавиши (или группы клавиш). В общем случае, нажатие клавиш в текущем состоянии либо оставляет калькулятор в этом же состоянии, либо переводит в новое состояние S{N}, для которого определен уже другой набор доступных для нажатия клавиш и, соответственно, другой список новых состояний, в которые калькулятор должен перейти при нажатии этих клавиш. То есть, в общем случае, нажатие клавиш переводит калькулятор между состояниями, для каждого из которых определен свой набор доступных для нажатия клавиш и свои переходы в новые состояния. Если в некотором состоянии S{N} нажата невалидная для этого состояния клавиша, то ввод просто игнорируется, оставляя калькулятор в текущем состоянии.

Для обеспечения ввода корректных чисел мы определили следующий граф состояний, где над каждым ребром, соединяющим два состояния S{A} и S{B} приведен список клавиш, нажатие которых переводит калькулятор из состояния S{A} в состояние S{B}.

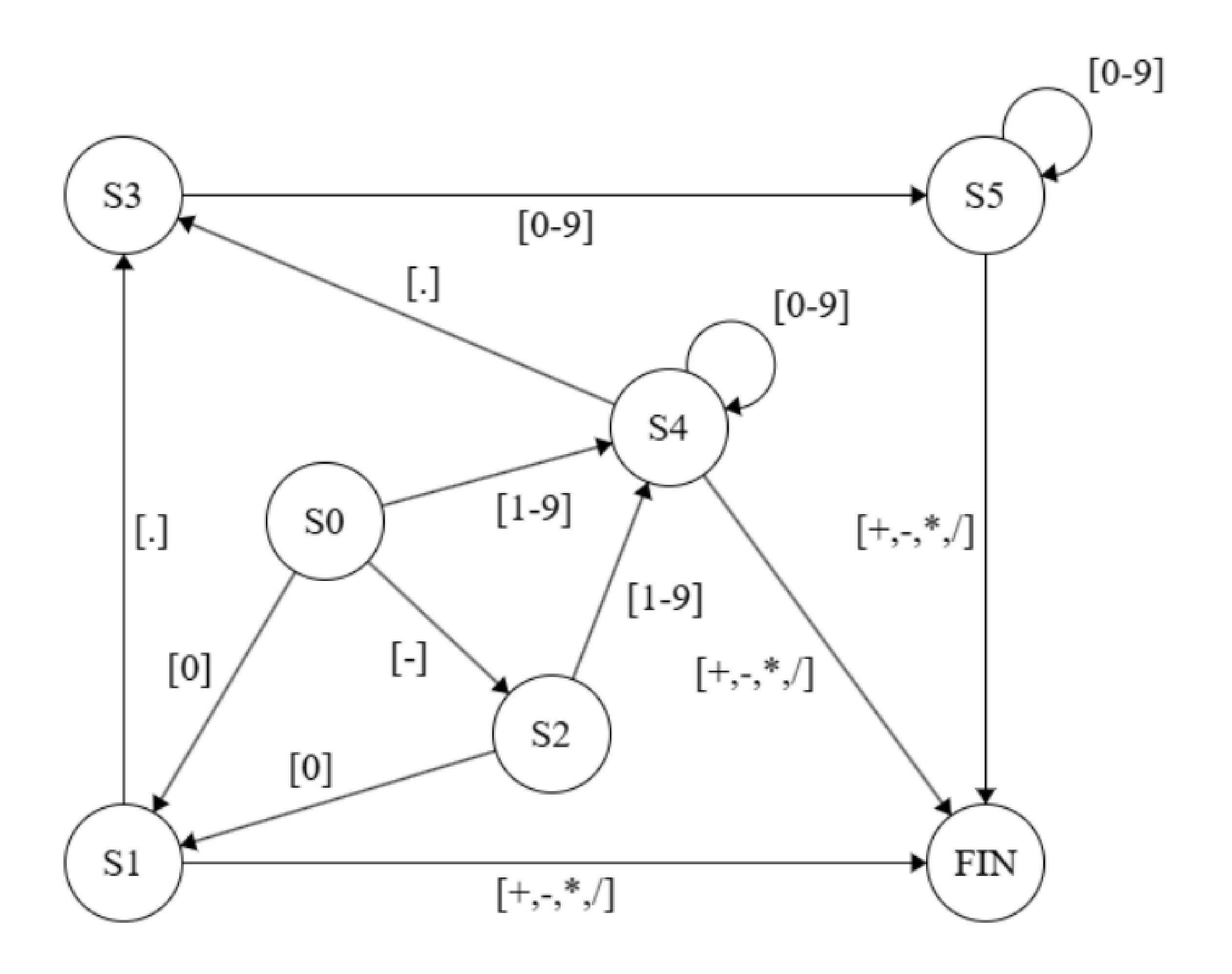


Рисунок 128

Реализован такой конечный автомат следующим образом: при нажатии клавиши, энкодер клавиатуры выставляет бит готовности данных и полубайтовый код нажатой клавиши. Бит готовности инициирует прерывание, по которому мы попадаем в обработчик прерываний по клавиатуре. Обработчик прерываний клавиатуры представляет собой набор подпрограмм, помеченных метками, каждая из которых реализует определенное состояние на графе состояний. В зависимости от текущего состояния, мы прыгаем на соответствующую метку.

При включении калькулятора в стеке находится адрес обработчика нулевого состояния, который расположен в коде по метке S0. При нажатии клавиши происходит переход на обработчик прерываний по клавиатуре, извлечение кода клавиши и конвертация его в ASCII-символ. Далее, из стека извлекается адрес обработчика текущего состояния, загружается в регистр Z и происходит прыжок на этот обработчик через вызов инструкции IJMP.

Допустим, что обработчик состояния S0 отработал (мы прочитали сырой код энкодера, смапились в ASCII-код, сохранили в SRAM введенный символ и вывели его на LCD). Также допустим, что при этом была нажата клавиша минус, которая переводит калькулятор в состояния S2, в котором клавиша минус уже не является доступной для нажатия. Как реализуется переход в новое состояние? До выхода из обработчика прерываний по клавиатуре, мы берем адрес метки S2 и помещаем в стек вместо адреса метки S0, который был там после первого включения калькулятора. После этого мы возвращаем управление из прерывания.

Теперь, если пользователь снова нажмет клавишу минус, мы снова попадем в обработчик прерываний, но поскольку в стеке уже находится адрес метки S2, мы прыгнем в коде не на метку S0, а на метку S2, где расположен обработчик состояния S2, который определяет уже новый набор клавиш доступных для нажатия, в который не входит символ минуса, и новый список состояний, в которые калькулятор перейдет после нажатия разрешенных клавиш. Таким образом, нажатие клавиши минус обработчиком S2 просто проигнорируется и калькулятор останется в состоянии S2 до тех пор, пока не будет нажата разрешенная в состоянии S2 клавиша и тем самым не произойдет изменение текущего состояния.

Вывод данных на дисплей

В калькуляторе используется дисплей LCD1602 на базе контроллера HD44780.

Детальная информация есть в даташите к LCD 1602 и контроллеру HD44780, а также — в коде библиотеки lcd1602.asm. Здесь же будут описаны только общие принципы взаимодействия.

Взаимодействие с контроллером LCD-дисплея

Линия управления состоит из 3 битов: RS, RW, E.

Бит RS определяет регистр в LCD, к которому мы обращаемся. RS=0 – регистр инструкций. RS=1 – регистр данных.

Бит RW определяет текущий режим: чтение или запись. RW=0 – запись. RW=1 – чтение.

Бит E – на нем мы формируем импульс определенной длительности (длительность определена в даташите) чтобы триггернуть LCD на выполнение работы.

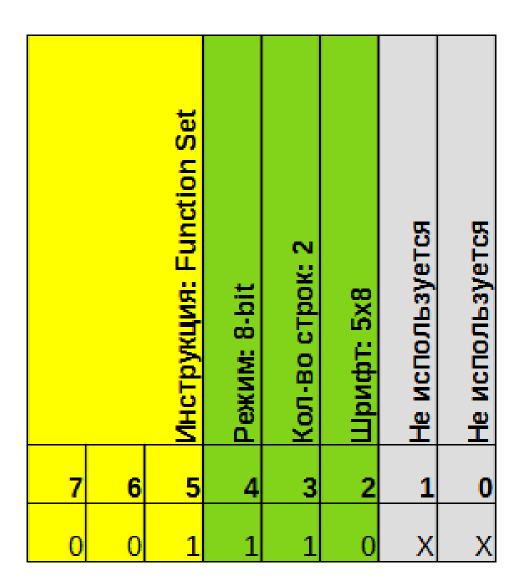
Линия данных — 8 бит (дисплей также может работать и в 4-битном режиме, принимая или возвращая данные по 4 бита последовательно).

Управление дисплеем происходит через передачу по линии данных кодов инструкций вместе с параметрами. При этом RS должен быть установлен в 0, что определяет регистр инструкций, а RW должен быть установлен в 0, что определяет запись.

Если мы попытаемся передать в дисплей следующую инструкцию, пока он занят обработкой текущей, то следующая инструкция будет проигнорирована. Поэтому для синхронизации с LCD существует т. н. busy-бит, который устанавливается контроллером дисплея в единицу если дисплей занят и сбрасывается в ноль, когда дисплей готов к обработке следующей инструкции. Для синхронизации часто используют другой подход: в даташите для каждой инструкции определено предельное время выполнения, следовательно, после отправки инструкции можно просто выжидать определенное время перед отправкой следующей.

У нас нет необходимости в экономии разрядов микроконтроллера, поэтому мы используем наиболее прямой и полный способ взаимодействия с дисплеем: 8-битная передача данных и синхронное ожидание сброса busy-флага. Для чтения busy-флага нужно сбросить бит RS в ноль и установить бит RW в единицу. Понятно, что порт B микроконтроллера (а в нашей реализации шина данных висит именно на этом порту) при этом нужно переключить на вход. После этого, контроллер дисплея выставит на 7-м бите (нумерация с нуля) шины данных значение busy-флага. Цикл чтения повторяется до тех пор, пока busy-флаг не окажется сброшенным.

Передача инструкций выполняется очень просто и однотипно. Рассмотрим на примере выполнения инструкции настройки дисплея на работу в 8-битном режиме с поддержкой двух строк и шрифта 5 на 8 точек:



Инструкция: Function Set			Режим: 8-bit	Кол-во строк: 2	Шрифт: 5x8	Не используется	Не используется
7	6	5	4	3	2	1	0
0	0	1	1	1	0	X	X

Рисунок 129

Опять же, поскольку мы выполняем инструкцию, то на линии управления должен быть выбран регистр инструкций и режим на запись: RS=0, RW=0. Когда код инструкции и параметры выставлены, генерируем импульс готовности E длительностью, которая указана в даташите (не менее 400ns).

Вывод одиночных символов

Для вывода символа достаточно выставить на шине данных ASCII-код символа и сформировать импульс E. При этом, в качестве ригстра назначения необходимо выбрать регистр данных: RS=1, а также выбрать режим записи данных: RW=0. Помним, что перед отправкой любых данных на дисплей мы дожидаемся сброса busy-флага, в коде — это подпрограмма WAITBUSY.

Вывод строки

Читаем строку из SRAM в цикле и выставляем посимвольно на шине данных ASCII-коды отдельных символов, дожидаясь WAITBUSY.

Управление курсором и очистка экрана

Управление курсором происходит через инструкцию Set DDRAM Address.

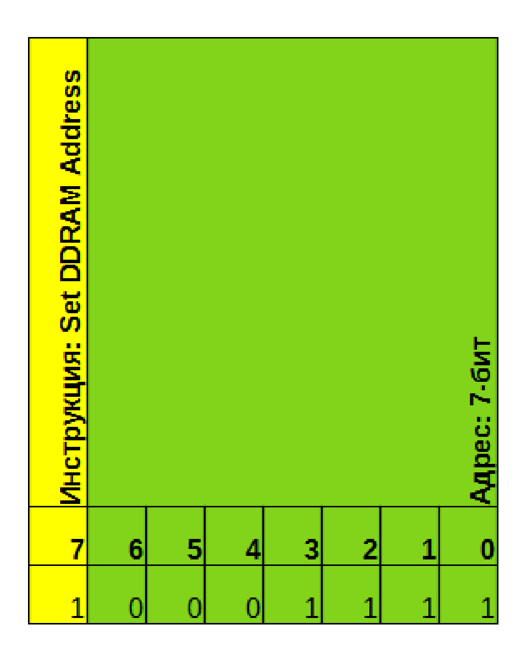


Рисунок 130

В режиме двух строк адреса знакомест первой строки находятся в отрезке [0x00,0x27], а адреса второй строки находятся в [0x40,0x67]. При этом видимых символов в каждой строке всего 16. Таким образом, адрес последнего видимого символа в первой строке равен 0x0F (0b0001111), а адрес последнего видимого символа во второй — 0x4F (0b1001111).

NOTE: Под адрес отведено 7 бит а не 8, т. к. 8-й отведен под код инструкции.

Аппаратная часть

Ниже приведена схема подключения компонентов, из которой должно быть ясно, какие компоненты используются, в каком количестве и как подключаются между собой.

Для питания используется 9-вольтовая крона.

Конденсаторы все неполярные, взяты прямо по даташитам.

Конденсаторы на 0.1uF и 1uF возле энкодера конфигурируют частоту опроса клавиатуры и задержку для внутренней схемы устранения дребезга контактов (детали см. в даташите на энкодер).

Резистор на 2K подключен к подсветке дисплея. На самом деле на плате дисплея уже, кажется, установлен резистор перед диодом подсветки, но дисплей довольно яркий, поэтому смело ставим еще один.

Контроллер работает от внутреннего генератора на 8MHz. То есть, при прошивке надо установить соответствующие фьюзы. Эта информация есть в даташите на МК, еще можно поискать калькулятор фьюзов (ссылки не привожу, поскольку они обычно недолговечны).

Клавиатуру нужно подключить так, чтобы при нажатии клавиш слева направо и снизу вверх (т. е. от «C» до «+») коды энкодера формировались от 0x00 до 0x0F, т. е., чтобы при нажатии «C» на выходе энкодера был 0x00, а при нажатии «+» — 0x0F. На самом деле, порядок подключения не так важен, он определяет только какие ASCII-коды находятся по полбутайтам энкодера, но наш код код прошивки написан с учетом именно такого подключения.

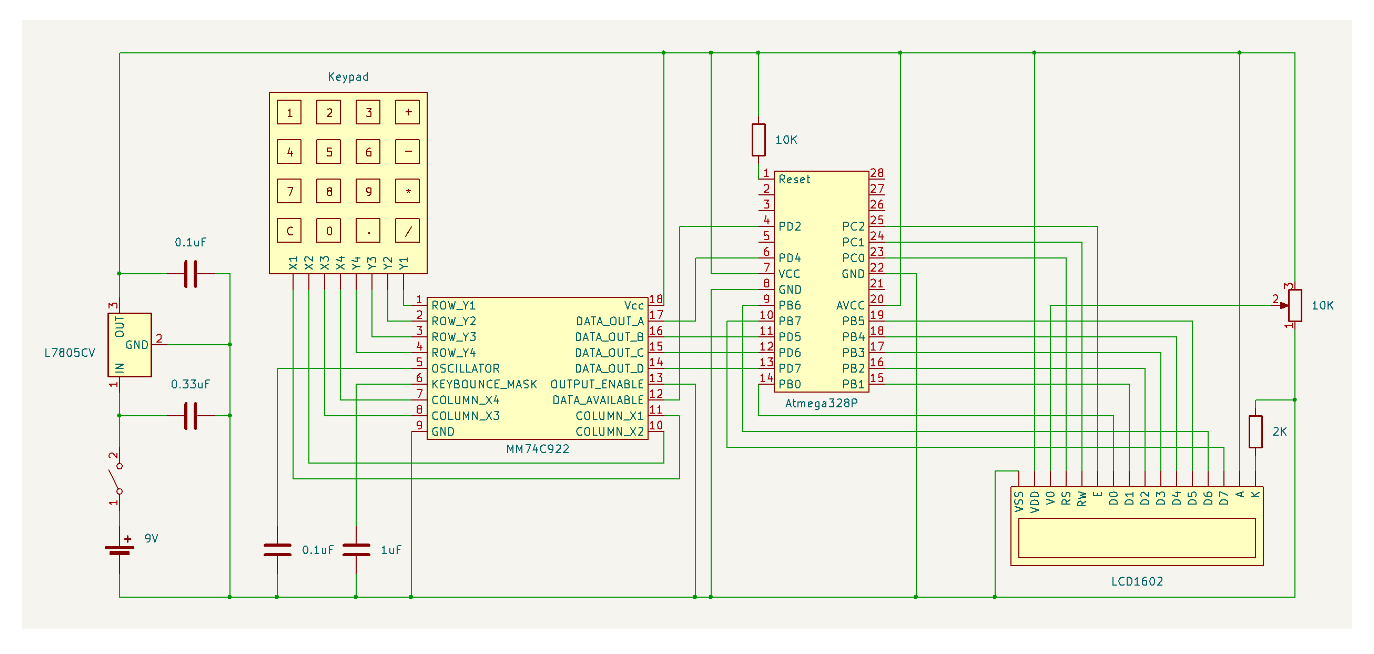


Рисунок 131